APPENDIX D

GCCS/JOPES CORE BACKUP/RECOVERY SOFTWARE

APPENDIX D - GCCS CORE BACKUP/RECOVERY SOFTWARE

GCCS Core backup/recovery software is designed to perform automated full, cumulative, and redo log backup/restore operations. The driving design issues include ease of use, online backup capability, and database recovery assurance. Figure D-1 illustrates the automated online backup capability. Maintenance of backup/recovery software will require in-depth experience with complex Bourne shell Unix scripts and SQL*Plus commands.

User procedures for backup/recovery programs are described in the *GCCS System Services Administrator's Manual*, Appendix C, dated 31 March 1997.

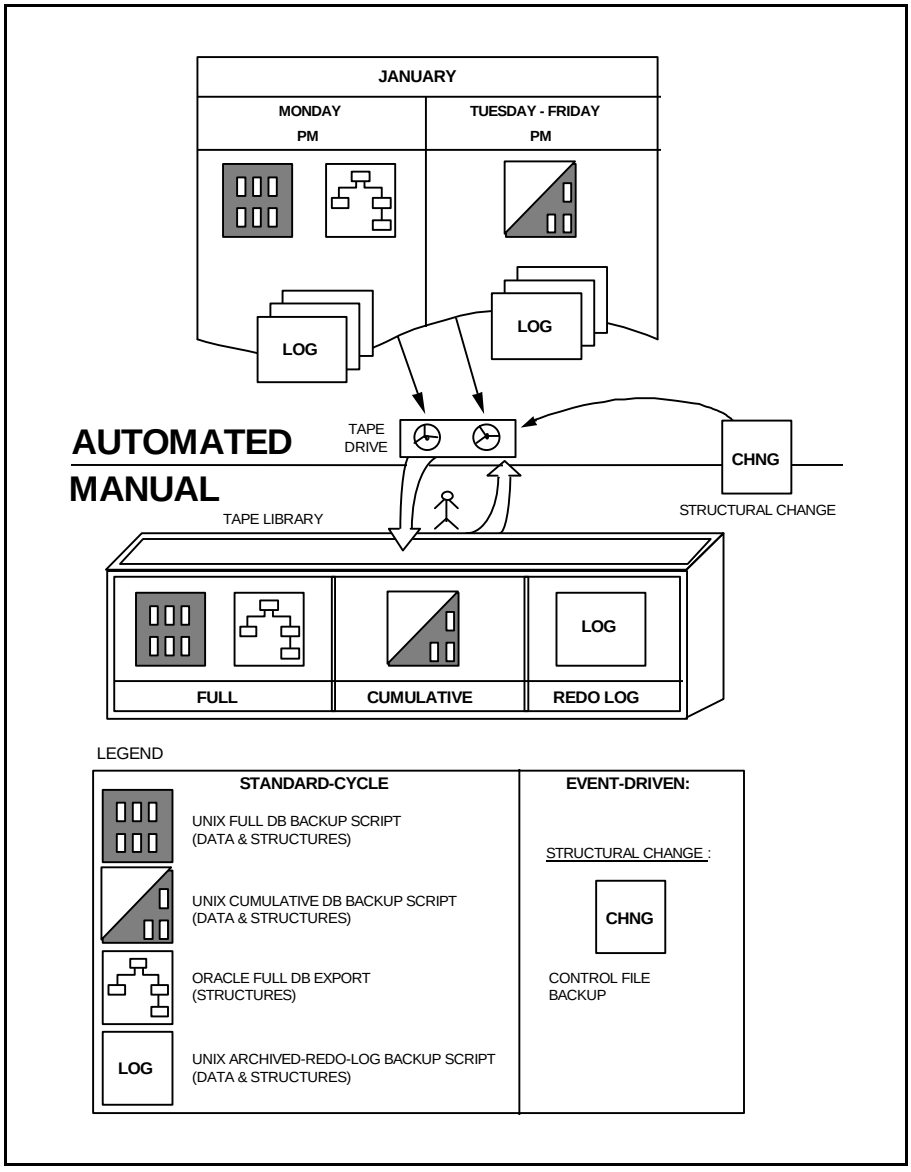

Figure D-1: Backup/Recovery Process.

D.1 SYSTEM REQUIREMENTS

Backup/recovery software maintenance requires a correctly configured server. Access to a dedicated 8mm or 4mm external backup tape drive is critical to backup/recovery development and testing.

The following backup/recovery requirements must be supported in the maintenance environment:

- SunOS (Solaris) Release 2.3 should be installed on the maintenance platform.

- ORACLE Version 7.1.3 should be installed on the maintenance platform.

- An externally identified <*oradba*> account must exist with **connect**, **resource,** and **dba** database privileges. A <*secman*> unix account should also exist.

- The <*oradba*> and <*secman*> user must be included as part of the **tty** group in the **/etc/group** file. This allows the backup program messages to be displayed on the console.

- The archive configuration parameters in **$ORACLE_HOME/dbs/configGCCS.ora** must be entered as follows:

  log_archive_dest = /oracle/smback/arch/GCCS
  log_archive_format = %_%s.log

  The **/GCCS** in the directory path is the database name or **$ORACLE_SID** and represents the name of the log file. This configuration will produce the following logs: (**GCCS_1.log, GCCS_2.log, GCCS_3.log**, etc.)

- All backup/recovery directories and files must be owned by the <*oradba*> user, **dba** group, and the directory structure cannot be changed when the software is installed. The top level directory must be **$ORACLE_HOME/RECOVERY** with all the subdirectories the same as the system that files were extracted from with all original unix file permissions intact.

- Online redo log archiving should be enabled and configured for a production site. A log file size of at least 10MB is recommended.

- All data files in the **$ORACLE_SID**=GCCS database must be readable and writable by the <*oradba*> user. The <*oradba*> user umask should default to read/write to ensure proper permissions when expanding tablespaces. To prevent inadvertent deletion of data files by other users, only users with access to the **dba** group should be permitted to delete data files.

- The contingency redo log backup functionality only searches for unused space on mount points owned by <*oracle*>. As of release patch SMDBP.6, only the archived redo log backup disk is owned by <*oracle*>, causing the contingency redo log backup functions to be inoperative.

D.2 ARCHITECTURE

The server backup/recovery software consists of automated programs executed from a character-based menu interface. The menu interface is started at the Unix prompt via the **br_main** startup script. The startup script is accessible from any current working directory because it is specified in the <*oradba*> user .login **path**. The **br_main** startup script calls the **br_main_menu** program which displays the screen shown in Figure D-2.

```
        B A C K U P  A N D  R E C O V E R Y  M E N U

    ONLINE BACKUP OPTIONS:      RECOVERY OPTIONS:

    (F)  FULL BACKUP            (RF)  RESTORE FULL BACKUP

    (C)  CUMULATIVE BACKUP      (RC)  RESTORE CUMULATIVE BACKUP

    (R)  REDO LOG BACKUP        (RR)  RESTORE REDO LOGS

    UTILITIES:

    (A)  CHANGE AUTOMATIC BACKUP SCHEDULE

    (B)  BACKUP STATUS

    (D)  DEVICE/DATABASE SETTINGS

                          (Q)  QUIT

        Please Select an Option and Press <ENTER>.
```

Figure D-2: Backup and Recovery Menu Screen.

The progress flow of each program is initiated when the program is selected from the BACKUP AND RECOVERY MENU. Figure D-3 shows the overall backup/recovery process architecture. The figure number for each program's architecture diagram is referenced in the overall backup/recovery process architecture diagram.

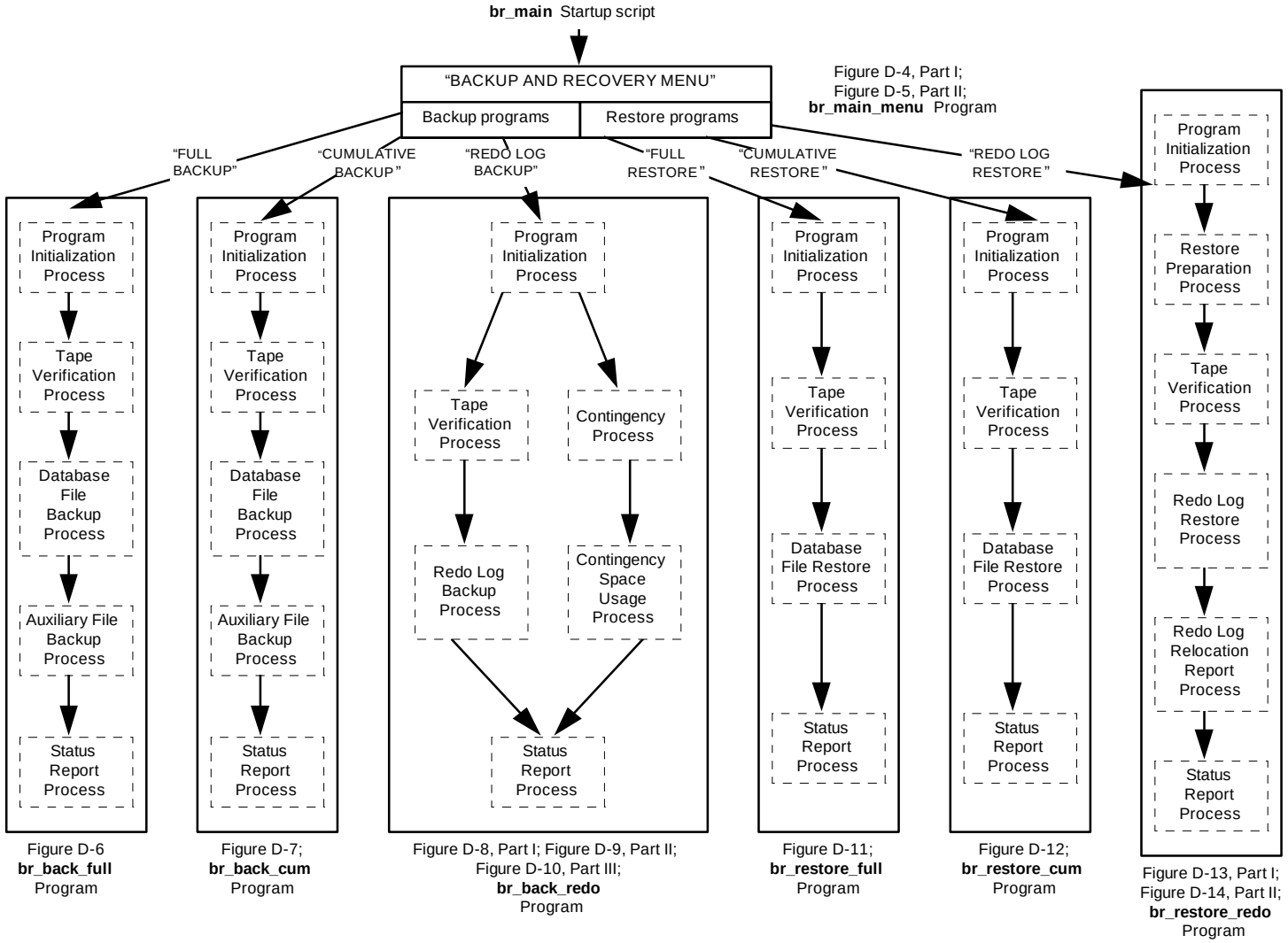

Figure D-3: Backup and Recovery Architecture Overview.

The composition of each program is illustrated by **process**, **module,** and **main body**:

- A **process** is a logical grouping of modules and represents a critical operation.

- A **module** is a collection of Unix commands which performs a task and is similar to the construct in the "C" programming language. A module has a name and is called when the task is required.

- The **main body** represents a collection of Unix commands executed in the main body of the code and performs a particular task. Generally, code in the main body performs a particular task once, while code in a module performs a particular task more than once.

Detailed design information for each program is provided in Paragraph D.3, Design. Process, module, and main body names are referenced explicitly as they appear in the architecture diagrams.

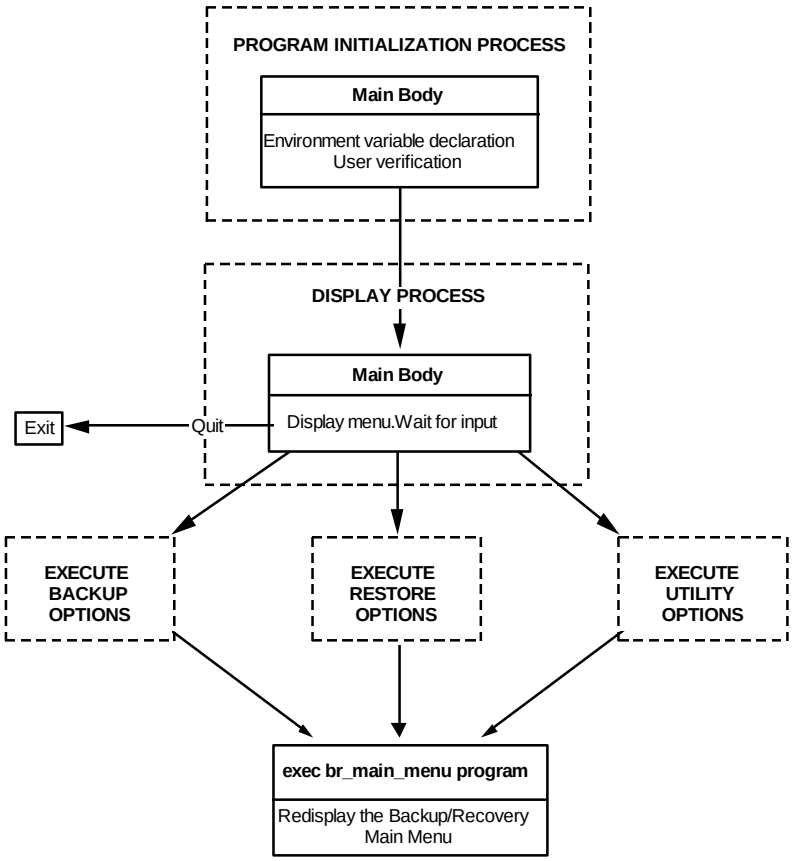

Figure D-4: ***br_main_menu*** *Program, Part I.*

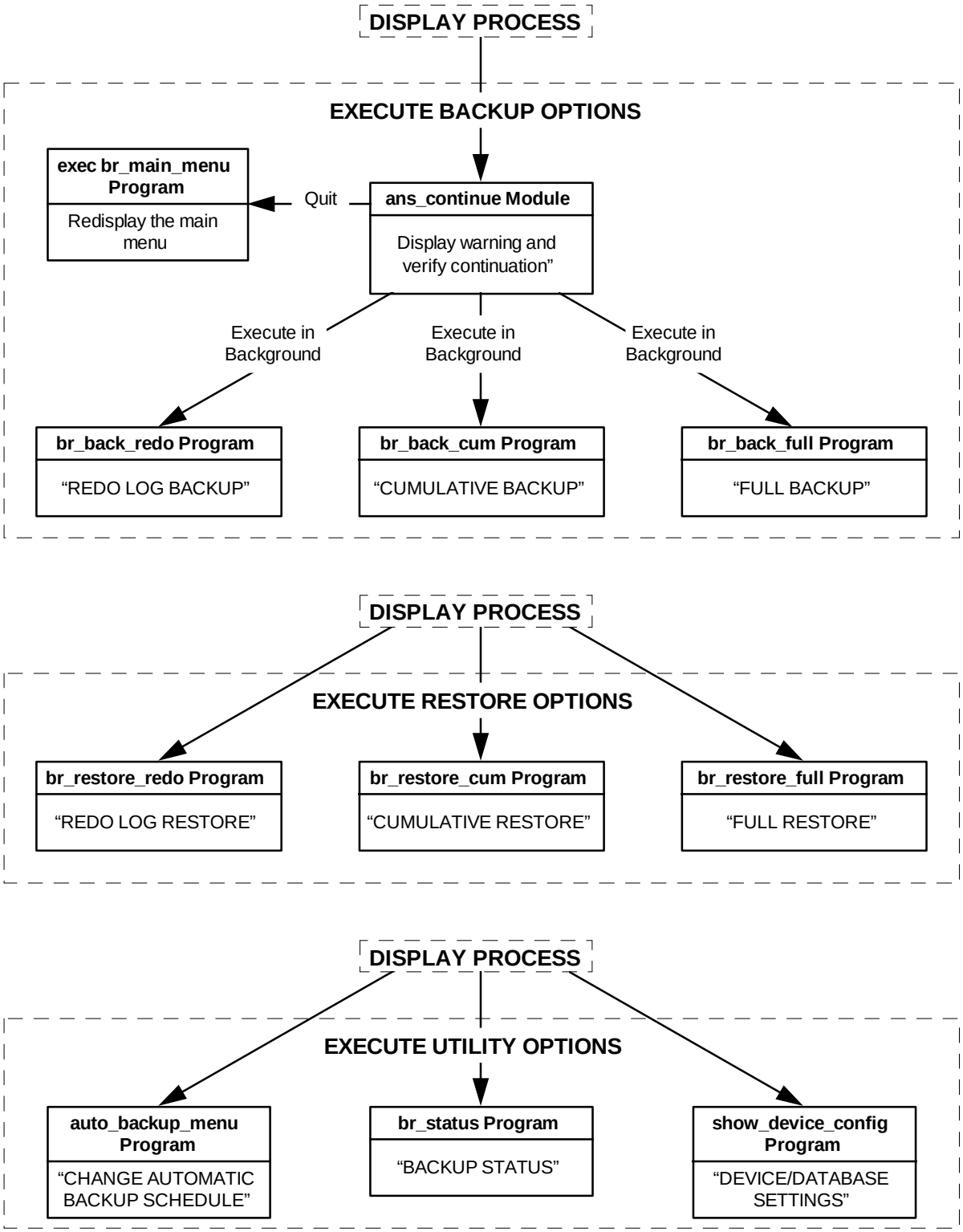

Figure D-5: ***br_main_menu*** *Program, Part II.*

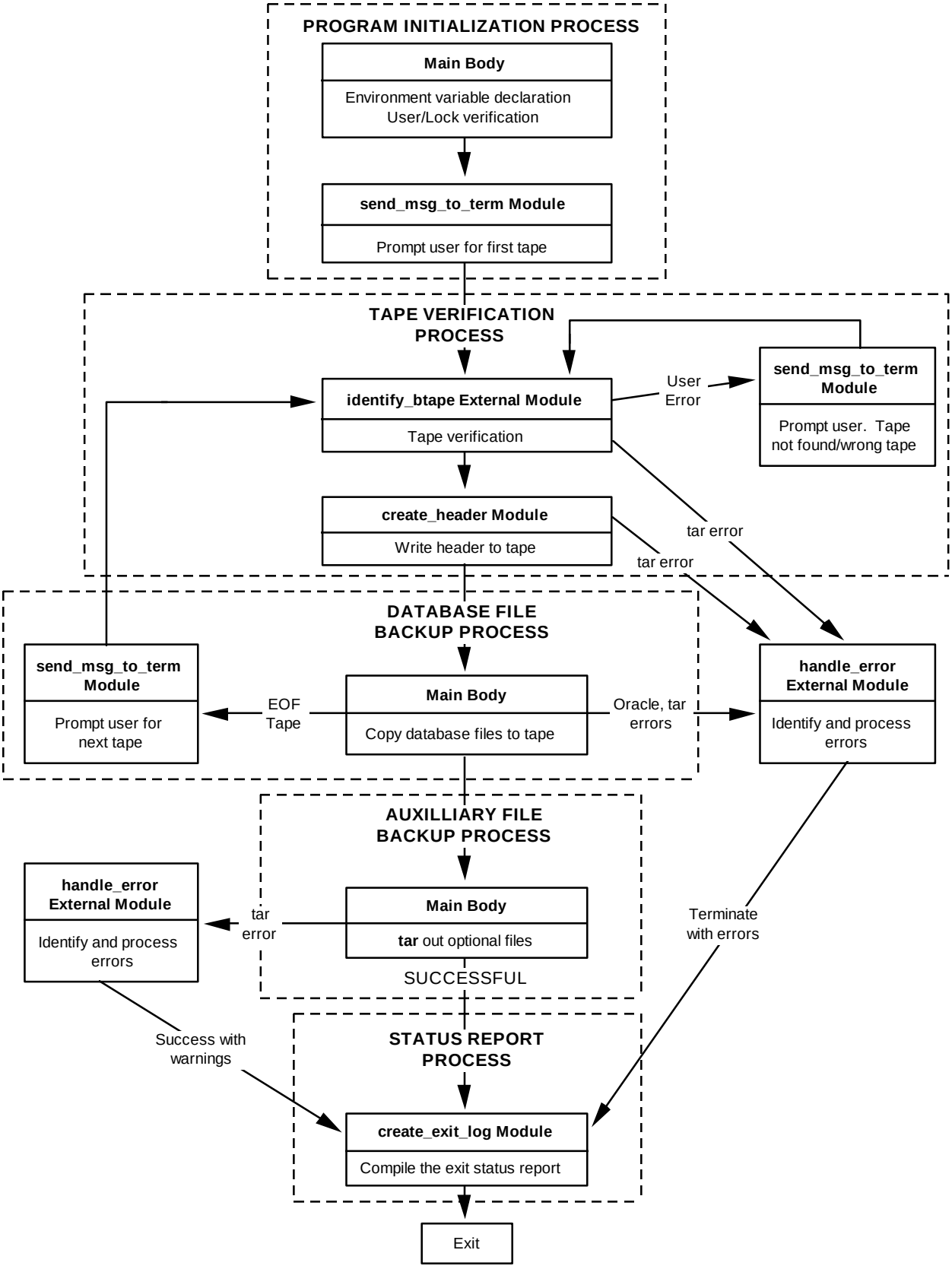

Figure D-6: ***br_back_full*** *Program.*

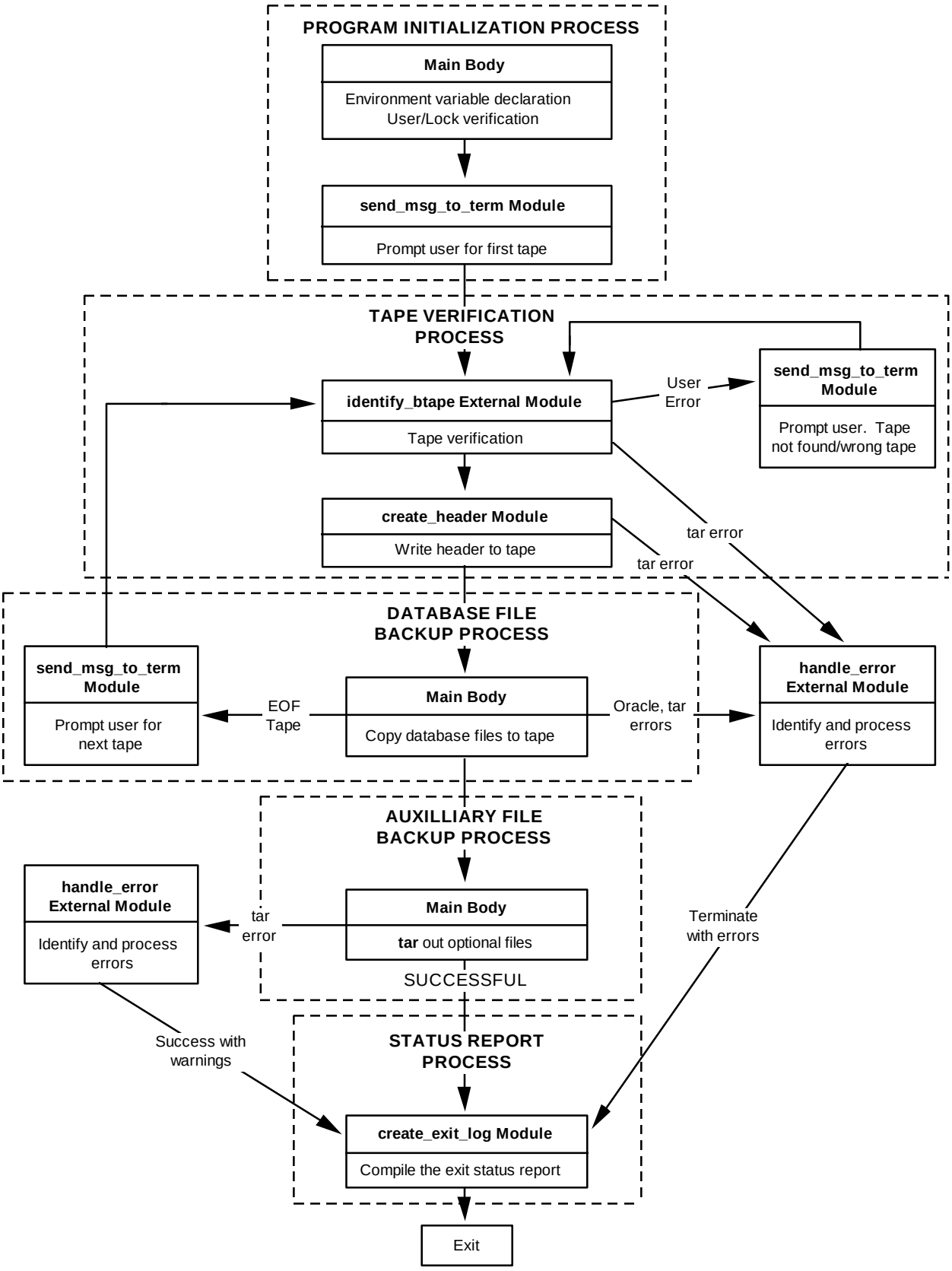

Figure D-7: ***br_back_cum*** *Program.*

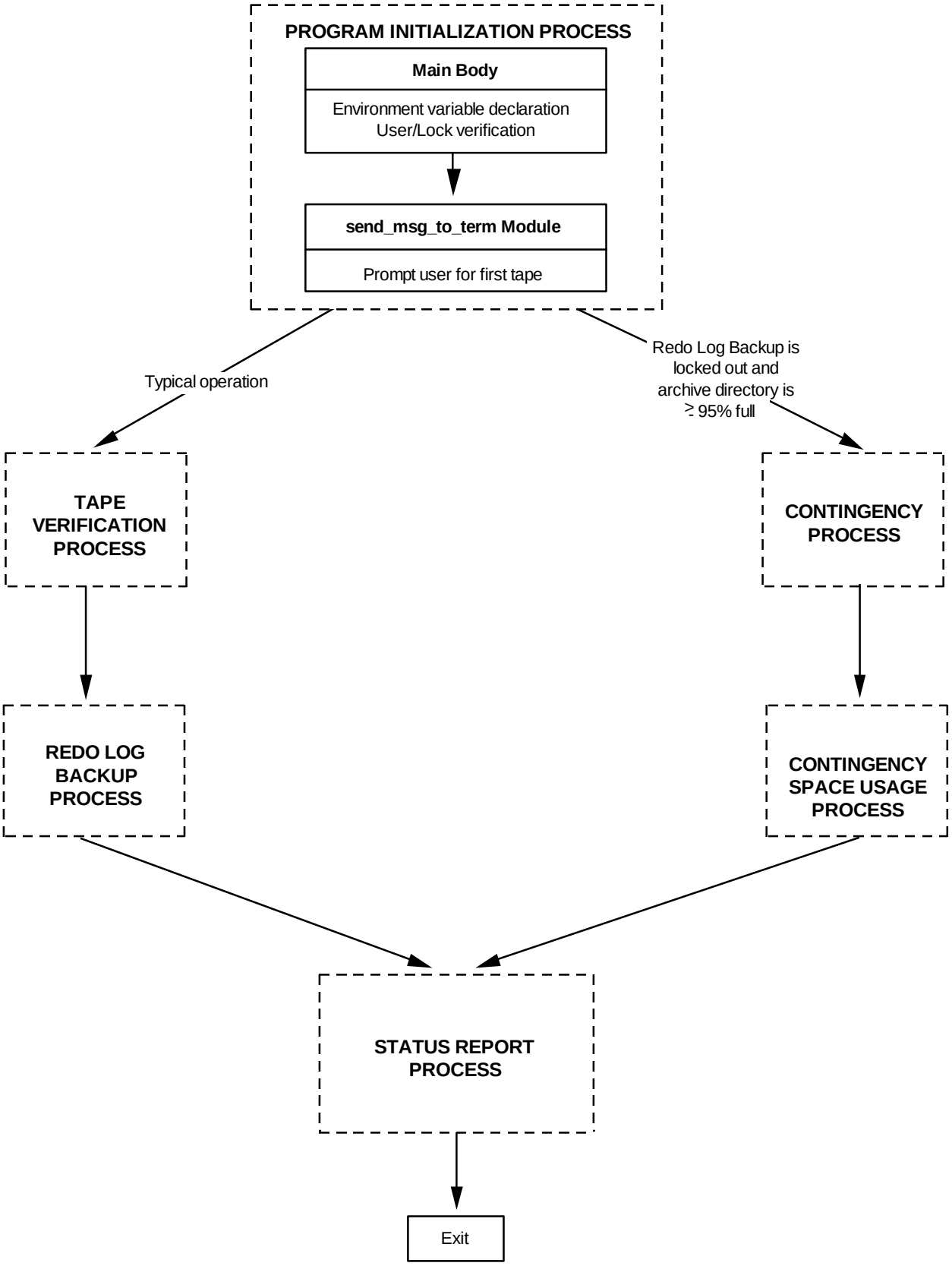

Figure D-8: ***br_back_redo*** *Program, Part I.*

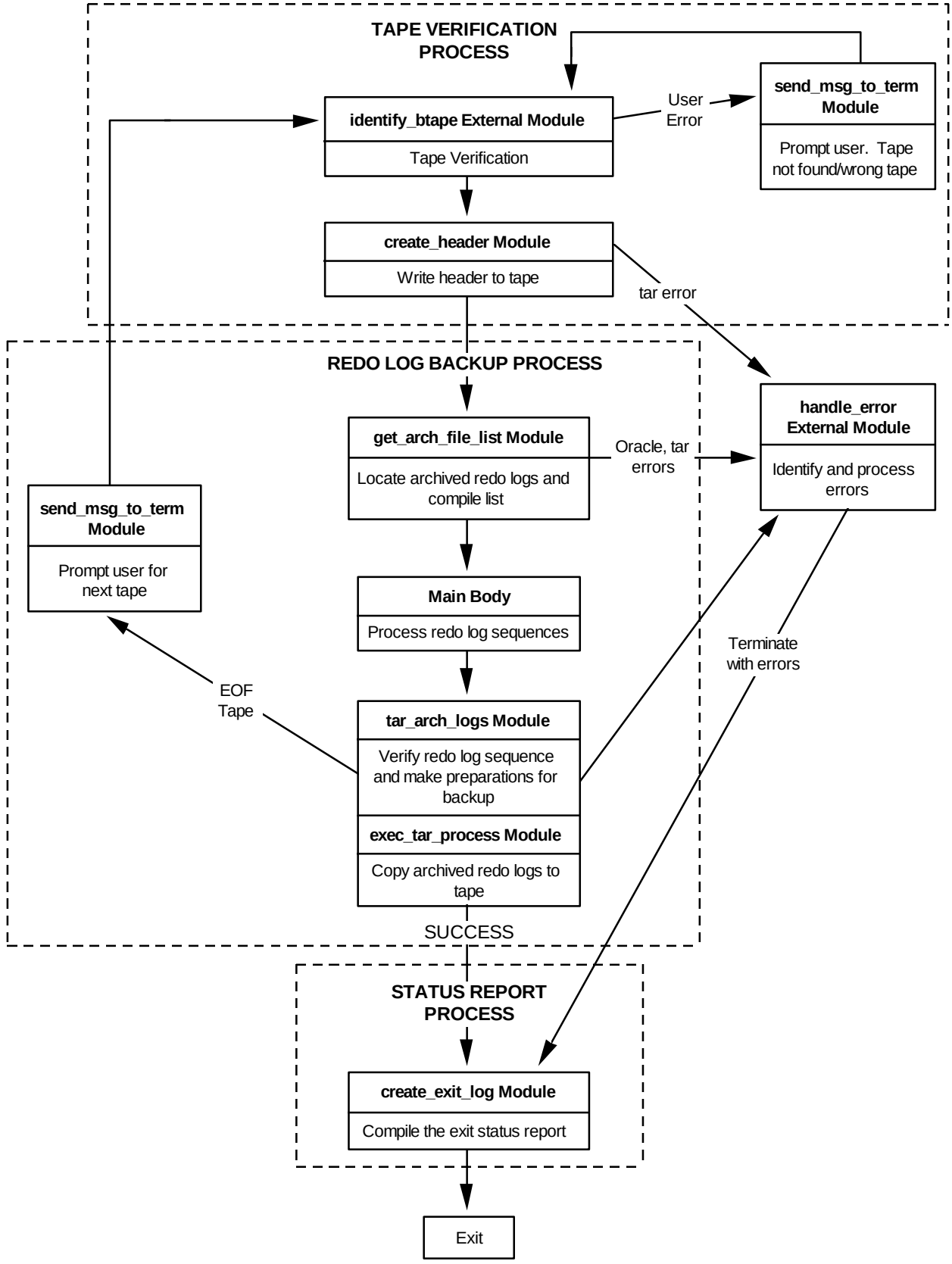

Figure D-9: ***br_back_redo*** *Program, Part II.*

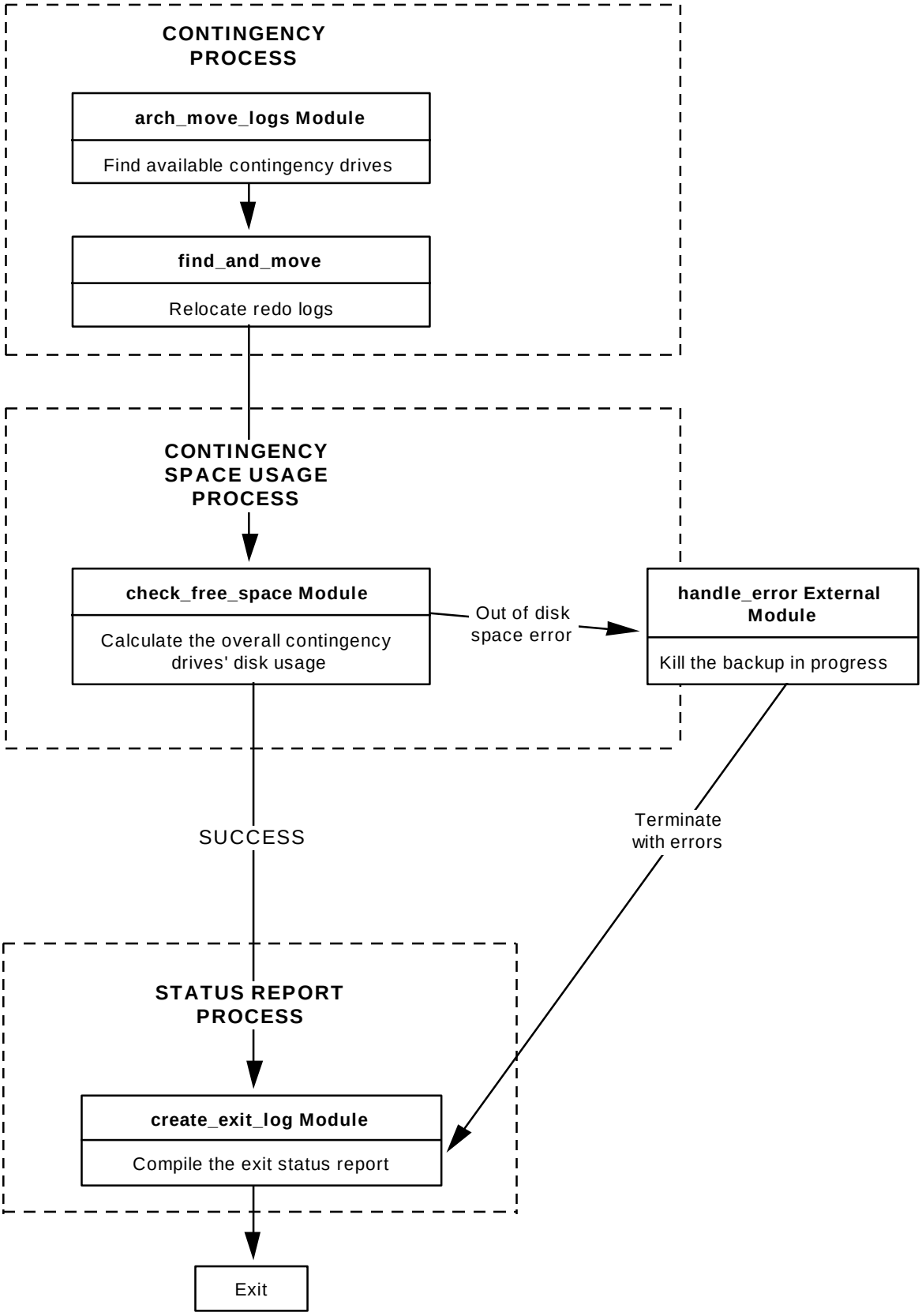

Figure D-10: ***br_back_redo*** *Program, Part III.*

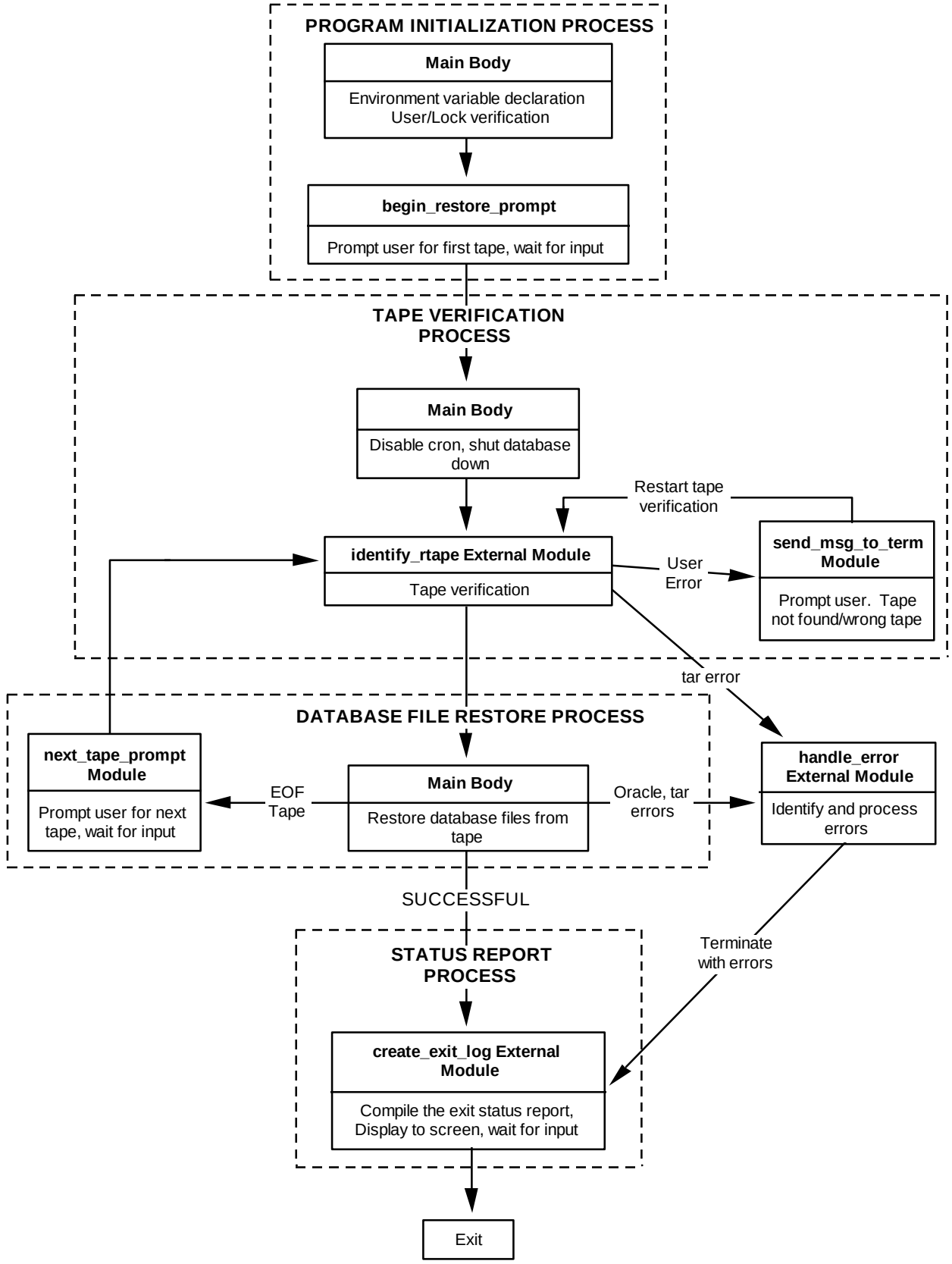

Figure D-11: ***br_restore_full*** *Program.*

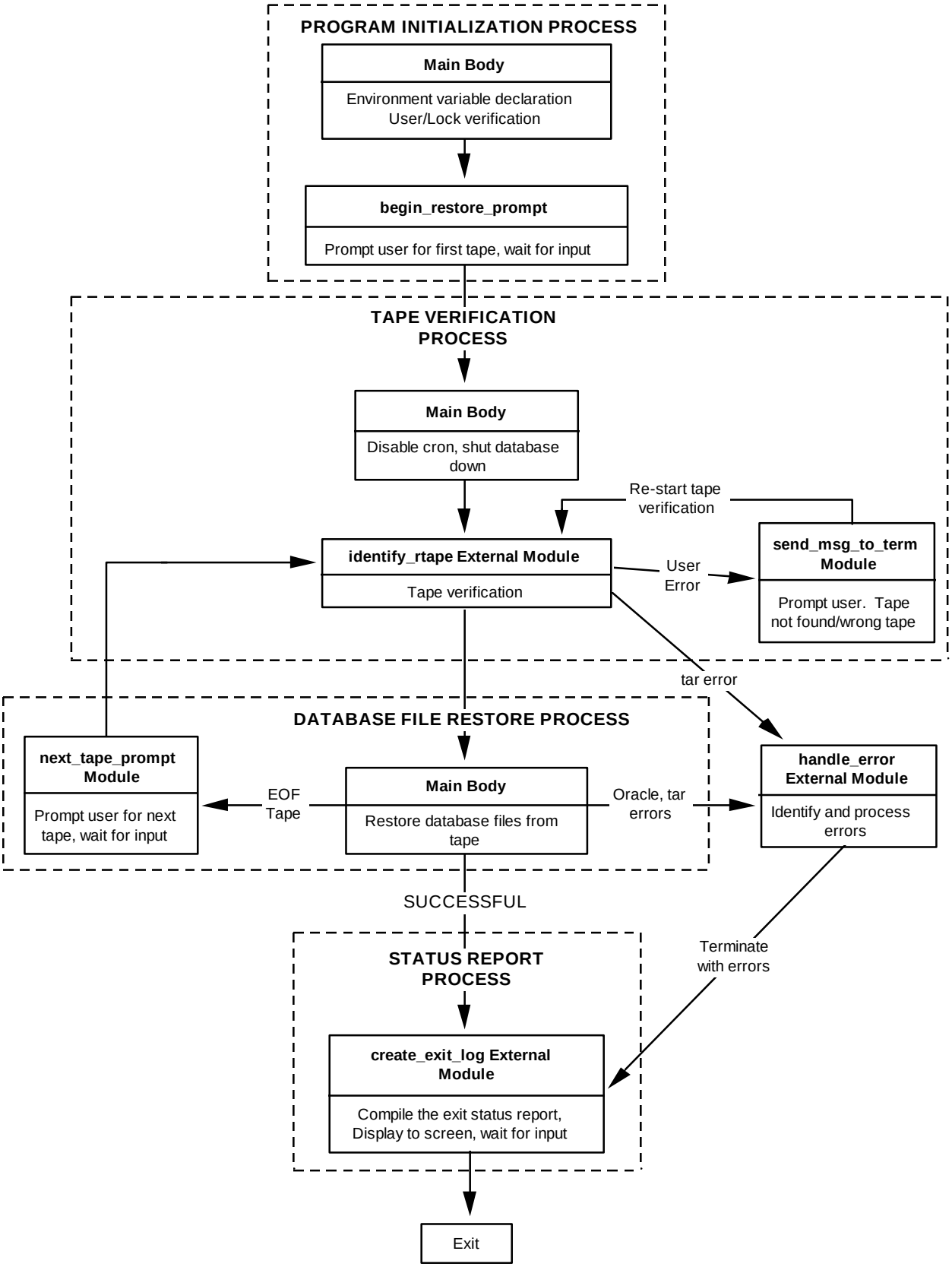

Figure D-12: ***br_restore_cum*** *Program.*

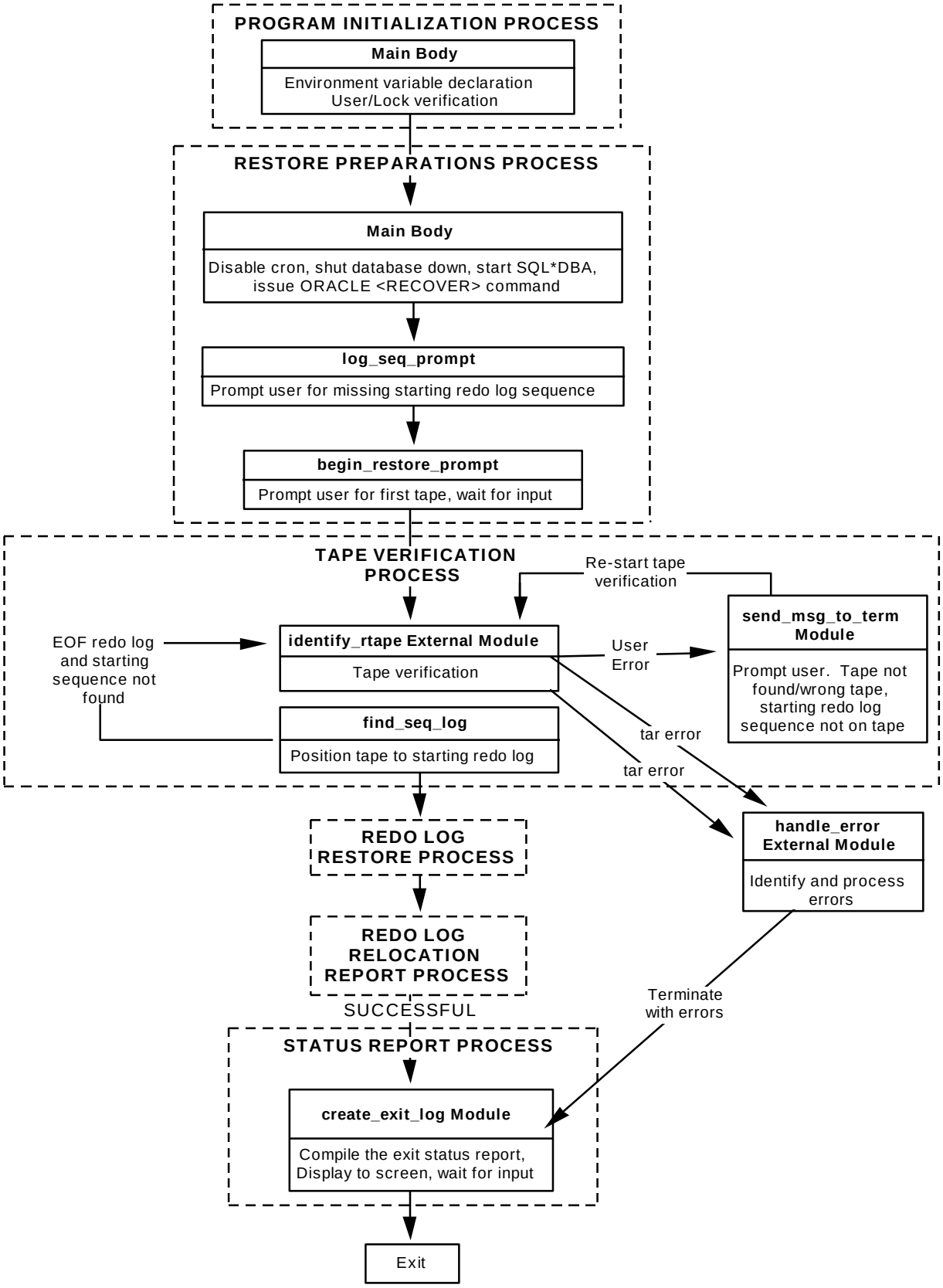

Figure D-13: ***br_restore_redo*** *Program, Part I.*

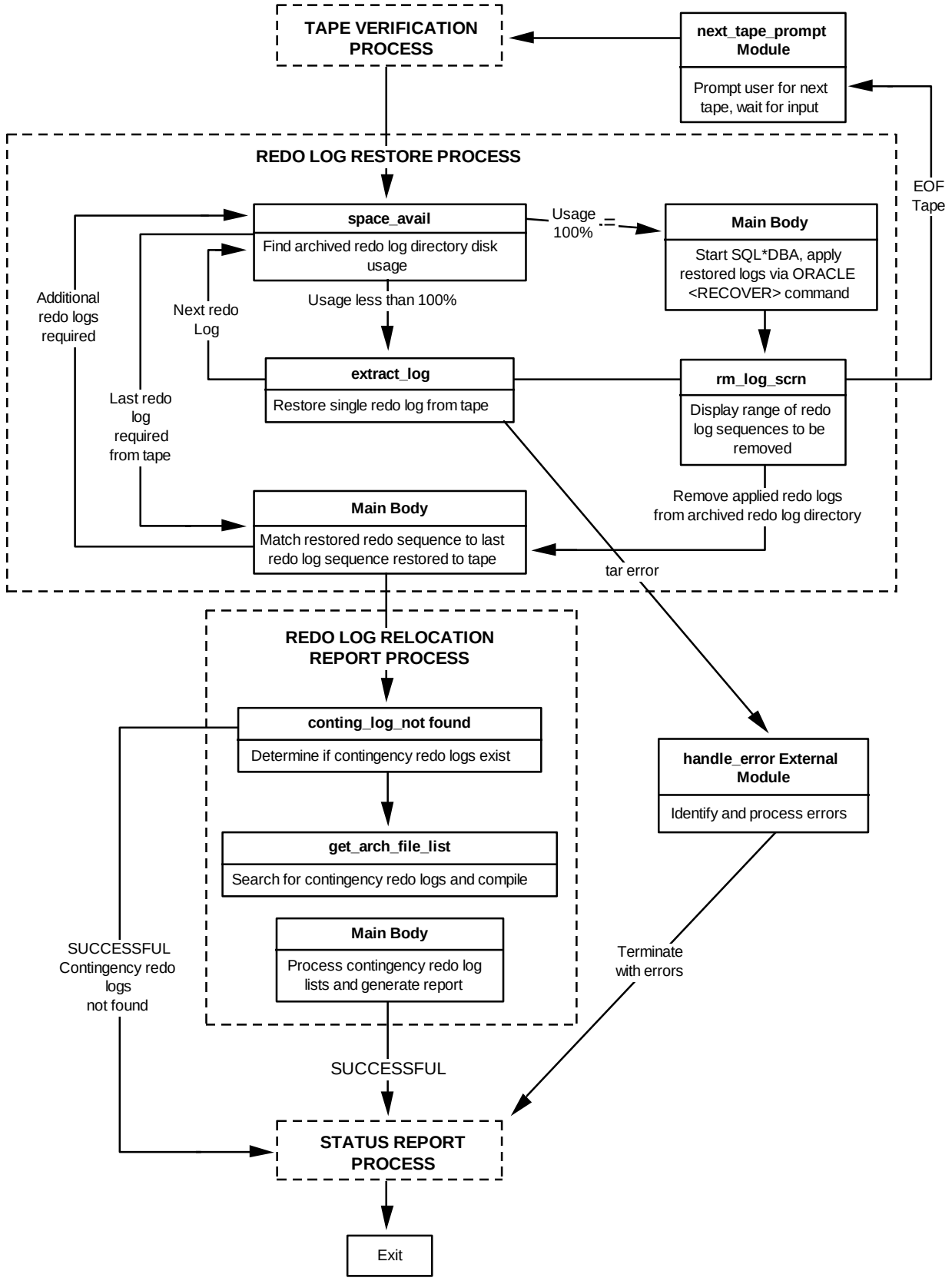

Figure D-14: ***br_restore_redo*** *Program, Part II.*

D.3 DESIGN

The design of each backup/recovery program is described in detail in this section. Architecture diagrams for each critical program are provided in Paragraph D.2, Architecture. Design information is provided for maintenance of the following programs:

- Backup/recovery main menu program. This program displays the menu interface (shown in Figure D-2) shared by the backup/restore programs.
- External error handler module. This stand-alone module performs error handling for the backup/restore programs that follow.
- Backup Programs. These programs consist of the full, cumulative, and redo log backup programs.
- Restore Programs. These programs consist of the full, cumulative, and redo log restore programs.

The backup/recovery architecture and design sections logically group the major backup or restore operations by processes within each program. A **process** is comprised of logical functions and a function is comprised of **main body** Unix commands and/or modules. A **module** is a collection of Unix commands which performs a certain task, generally more than once. For example:

- The full backup program logically consists of Program Initialization, Tape Verification, Database File Backup, Auxiliary File Backup, and Status Report processes. Processes are identified explicitly (i.e., Paragraph D.3.3.1, Program Initialization Process).
- The Database File Backup Process logically consists of Identifying Database Files, Copying Database Files to Tape, and Processing Multitape Backups functions. Each function is performed by main body and/or module scripts. Module scripts are identified explicitly (i.e., Paragraph D.3.3.3.2., copying database files to tape/**end_ts_backup** Module).

Logical groupings are used within this document to clearly represent the function of complex Unix scripts. They provide a valuable tool for the maintenance developer. For example, an update may be required to the portion of the full backup code which copies files to tape. The maintenance developer can use this document to identify the existing code within the backup process and determine which related processes may be affected.

D.3.1 Backup/Recovery Main Menu Program — br_main_menu

The **br_main_menu** program displays the menu of backup, restore, and utility options. It is designed to evaluate user input and execute appropriate backup, restore, or utility options as requested by the user. The restore and utility options are executed in foreground and require user

interaction. The backup options are executed in background and do not require direct user interaction. When a backup option is selected the program is executed and then immediately returns to the main menu.

The selection of a backup option invokes the call to a display screen which forewarns the user that a backup has been selected. The user has the option to continue or quit. This is a precautionary measure to prevent the user from accidently running a backup program. The restore or utility options do not incorporate the warning when selected.

The **br_main_menu** program is comprised of three processes:

- Program Initialization Process, Paragraph D.3.1.1
- Display Process, Paragraph D.3.1.2
- Program Execution Process, Paragraph D.3.1.3.

GENERAL ERROR HANDLING:

The **br_main_menu** program handles two types of error conditions, a login error, and an environment variable error. A login error occurs when a user does not login as <*oradba*> and tries to run the **br_main_program**. An environment variable error occurs when the *ORACLE_SID*, *TAPE* or *ORACLE_HOME* variables remain null after the program tries to derive the values. The program cannot continue if the login or environment error occurs. When the **br_main_menu** program detects error conditions, it displays the appropriate error message, waits for user response, and exits.

D.3.1.1 Program Initialization Process. The program initialization process declares and exports environment variables for the restore and utility programs. The backup programs have the declaration and export functions built into each program. This is required because the backup uses the **cron** facility which uses only environment variables declared in the program. The program initialization process performs preliminary checking. (See Program Initialization Process for details, Paragraph D.3.3.1).

D.3.1.2 Display Process. The Display process creates a standard menu on the screen and prompts the user for input.

D.3.1.3 Program Execution Process. The program execution process maps the following variable to the following command lines:

- *F* <nohup **/backup/full/br_back_full &**>
- *C* <nohup **/backup/cum/br_back_cum &**>
- *R* <nohup **/backup/redo_log/br_back_redo &**>
- *RF* <**/restore/full/br_restore_full**>
- *RC* <**/restore/cum/br_restore_cum**>
- *RR* <**/restore/redo_log/br_restore_redo**>

- *A* </**etc/auto_sched_menu**>
- *B* </**etc/br_status**>
- *D* </**etc/show_device_config**>

D.3.2 External Error Handler Module — handle_error

The handle_error module is designed to perform central error processing for both the backup and restore programs. The purpose for the design is to make the updates, additions, or deletions of error messages an easier task. If a new error condition needs to be handled, append the condition and message to the **handle_error** module and then place the **handle_error** call with the proper arguments in main program.

ARGUMENTS:

- First Argument — Identifies the type of error.
- Second Argument — The directory path of the application with the error condition. It is used to find the **error.log** and **current_progress** file.
- Third Argument — The application name in which the error occurred. This will be used as part of the error message.
- Additional Arguments — Used to further process the error condition.

The **handle_error** module evaluates the first argument from the calling program and writes the appropriate message to both the **error.log** and **current_progress** files and then returns. It implements a *large case* statement. Table D-1 maps the error argument to the description and calling program(s).

*Table D-1: External Error Handler Module — **handle_error** Arguments.*

ERROR ARGUMENT	DESCRIPTION	CALLING PROGRAM(S)
LOCK_ERROR	A backup is in progress.	**br_back_full** **br_back_cum** **br_back_redo** **br_restore_full** **br_restore_cum** **br_restore_redo**
IO_ERROR	Fatal error while trying to create a file.	**br_back_full** **br_back_cum** **br_back_redo**

ERROR ARGUMENT	DESCRIPTION	CALLING PROGRAM(S)
ORACLE	General ORACLE error.	**br_back_full** **br_back_cum** **br_back_redo** **br_restore_full** **br_restore_cum** **br_restore_redo**
ORACLE2	ORACLE error pertaining to the online backup process.	**br_back_full** **br_back_cum**
ORACLE3	ORACLE error pertaining to backing up of the **control** file.	**br_back_full** **br_back_cum**
NON_TERM_ERROR	Unix errors occurring as a result of backing up optional files.	**br_back_full** **br_back_cum**
LOGIN_ERROR	An attempt was made to perform a backup, restore, or utility option by a user not logged in as unix user <*oradba*>.	**br_back_full** **br_back_cum** **br_back_redo** **br_restore_full** **br_restore_cum** **br_restore_redo**
TAR	Fatal *tar* or tape drive error.	**br_back_full** **br_back_cum** **br_back_redo** **br_restore_full** **br_restore_cum** **br_restore_redo**
OUT_OF_SPACE	**br_back_redo** program is locked out and the contingency drive is out of space.	**br_back_redo**
SEQ_NOT_FOUND	The current log sequence could not be obtained from the database.	**br_back_redo**
VFILESTAT_ERR	A reliable **vfilestat** file could not be created.	**br_back_full** **br_back_cum**
TIMED_OUT	The appropriate backup tape was not inserted. System timed out after approximately three hours.	**br_back_full** **br_back_cum** **br_back_redo**
HOME_NOT_FOUND	The *dbhome* command was not found or it could not return the ORACLE home directory.	**br_back_full** **br_back_cum** **br_back_redo**
DEV_NOT_FOUND	A tape device was not defined in the **DEFAULT_TAPE** file. This value is assigned to the *TAPE* environment variable.	**br_back_full** **br_back_cum** **br_back_redo**

D.3.3 Full Backup Program— br_back_full

The **br_back_full** program performs an online full database backup. All datafiles of each tablespace are backed up while the database is available for updates and all normal operations. The

commands and modules which comprise the full backup program are grouped together to form the following processes:

- Program Initialization Process. Handles program execution/termination, user/program interaction, and preliminary checking.

- Tape Verification Process. Performs identification of full backup tapes, enforces the correct type of tape, and the correct sequence **(FULL_1, FULL_2**, etc.).

- Database File Backup Process. Locates the name of each tablespace and its corresponding database files from the **DBA_DATA_FILES** table and copies those files to tape.

- Auxiliary File Backup Process. Copies the backup **control** and **DB structure export** file to tape and spools out database file Input/Output (I/O) information from the **V$FILESTAT** table. The I/O information is a snapshot of the number of blocks written to each tablespace file and is used in conjunction with the cumulative backup program. **br_back_cum** uses the snapshot to determine which tablespaces have been modified since the last full backup.

- Status Report Process. Compiles and prints a backup summary log which contains the backup status (successful, successful with warnings or terminated with the error message). Also, if the program was successful, it contains the database files copied to tape grouped by tablespace name. The file is printed out at the end of the backup so the user can place the printout in the backup binder.

INPUT TABLES:

- **SYS.DBA_DATA_FILES**. Provides the tablespace names and database file names.

- **SYS.V$FILESTAT**. Provides the number of physical blocks written to each tablespace.

OUTPUT FILES:

- **vfilestat** — contains the spooled out **V$FILESTAT** information.

- **current_progress** — gives the user the ability to check the progress of the program. The file contains messages which describe all program events which processed at the time the file **current_progress** was called. The **current_progress** will also contain any program errors.

- **exit_status** — is a copy of the status report which contains the status of the program (successful, successful with warnings, or terminated with the error message) and the database files copied to tape grouped by tablespace name.

- **error.log** — is an accumulation of all error messages.

- **control.bak** — is the backup of the ORACLE **control** file.

TEMPORARY FILES:

- **/tmp/spooled** — raw output from the **SYS.DBA_DATA_FILES** table.

- **/tmp/input** — contains the tablespace names and database file names derived from the spooled file. This file is the main driver of the program.

- **/tmp/error_flag$$** — this file holds any ORACLE errors that are processed by the error handler.

- **/tmp/LK_BACKUP_FULL$$** — created when the program first starts and locks out any other backup or restore program from executing. The file is removed when the program terminates via the *trap* command and is also remove from the **/tmp** directory automatically if the system is rebooted.

- **/tmp/TABLESPACE_NAME** (e.g., **/tmp/CARRIER**) — contains the database file(s) which makes up a particular tablespace. Each tablespace selected for backup will have a corresponding **TABLESPACE_NAME** file created. For instance, **CARRIER** tablespace will have a file named **CARRIER** containing **/oracle1/carrier1.dbf** and so on.

GENERAL ERROR HANDLING:

All routine errors are trapped by the **handle_error** external module including write/permission errors, lock errors (another backup/restore program is currently running), *tar* I/O errors, and login errors. Refer to Paragraph D.3.2, External Error Handle Module — **handle_error**, for more details.

When any of the above errors occur the program writes the appropriate error message to the **error_log**, **current_status**, and **exit_status** files, then the error message is mailed to the <*oradba*> user account.

If an error occurs while backing up the **control file** or while the **V$FILESTAT** table is being accessed, the program will terminate successfully with a warning. As this warning implies, there is no need to start over because all the files are already (successfully) backed up to tape.

D.3.3.1 Program Initialization Process. The program initialization process is designed to perform the following functions:

- Program Execution, Paragraph D.3.3.1.1
- Program Termination, Paragraph D.3.3.1.2
- Monitoring Current Status, Paragraph D.3.3.1.3
- User Prompting/send_msg_to_term Module, Paragraph D.3.3.1.4
- Environment Variable Declaration, Paragraph D.3.3.1.5
- Preliminary Checking, Paragraph D.3.3.1.6
- Exit Handling, Paragraph D.3.3.1.7.

D.3.3.1.1 Program Execution. The program execution function provides the users with two mechanisms for starting the full backup:

- BACKUP AND RECOVERY MENU screen
- **cron** file (via AUTO BACKUP MENU screen).

Both the menu option and **cron** file execute the backup programs in background. The menu option places the program in background by using the *nohup* command in conjunction with the &. As a result the user has immediate control of the terminal after the backup option is selected and can then log off the terminal or continue working with no affect on the backup program.

The **cron** file automates the backup process by allowing the user to schedule backups. By default the **cron** file executes the backup program in background.

D.3.3.1.2 Program Termination. The termination function gives the user the ability to stop the full backup at any time. Program termination is initiated via the TERMINATE FULL BACKUP option from the BACKUP STATUS MENU screen.

D.3.3.1.3 Monitoring Current Status. The function of monitoring the backup status provides the user with an interface to view the history of backup activities. An interface is required because the backup program is a separate process running in background. The backup status is determined by the contents of the **current_progress** file. The **current_progress** file contains a history of backup activities including user prompts and any errors messages. The backup program appends entries into the **current_progress** file describing each stage of the full backup process. The user monitors the backup progress by reading the **current_progress** file via the BACKUP STATUS option from the BACKUP AND RECOVERY MENU screen. The current status will include all activities up to the point in time that the BACKUP STATUS option was selected. In order to view the status of continuing activities the user must exit the backup status display screen and reselect the FULL BACKUP STATUS option.

D.3.3.1.4 User Prompting/send_msg_to_term Module. The user prompting function is designed to communicate a request for a certain user action to take place before the program can continue. The message prompts must accommodate the application running as a background process. The backup program is incapable of suspending the current backup activity (in order to prompt the user

and wait for a response) because it cannot receive input from a keyboard. The message handling capability is accomplished by sending automated message prompts to the console and to all other terminals where <*oradba*> or <*secman*> users are logged in. The application senses a correct user response by monitoring the tape drive status. A loop construct is used to display messages and monitor tape drive activity.

Screen message handling is accomplished via the **send_msg_to_term** module. The module simply sends the last four lines in the **current_progress** file to console then to each user logged in as the unix user <*oradba*> or <*secman*>. Before the **send_msg_term_module** is called, the backup program writes the message to the **current_progress** files.

MESSAGE DESCRIPTIONS:

- The backup has just started and reminds user with the following messages:

  "Please make sure that the FULL BACKUP tape FULL_1 is in place."

- If the backup is in progress and requires an additional tape to continue, the following message will be displayed:

  "PLEASE REMOVE THE TAPE AND INSERT NEXT FULL BACKUP TAPE: FULL_2."

- If the program has identified an invalid tape, then the following messages are displayed:

  "The Full Backup cannot continue !!!!."

  "Please insert the FULL_1 tape, a new tape, or an erased tape."

- If the user forgot to put the tape in the drive or specified the wrong tape device, the following message is displayed:

  "THE FULL_1 TAPE IS NOT LOADED OR THE TAPE DRIVE IS OFFLINE !!!."
  "Please insert the tape labeled FULL_1 or confirm that the device driver is installed properly."

D.3.3.1.5 Environment Variable Declaration. A critical design issue involves deriving values for the *TAPE* and *ORACLE_SID* environment variables and printout destination variables from a source outside of the **br_back_full** program. This provides the means to implement user configurable environment variables. In addition, it also prevents the bad practice of hard coding the values into the application code. The *TAPE* and *ORACLE_SID* variables are essential to the backup process while the printout destination variable *PRINTER* is optional.

The *ORACLE_SID*, *DEFAULT_TAPE*, and *PRINTER* values are each stored in a separate file under **$ORACLE_HOME/RECOVERY/etc**. The application, during program initialization, determines the path and assigns the values which are then exported. Variable settings are provided in Table D-2.

.

Table D-2: Environment Variable Declaration Settings.

SETTING	SOURCE FILE AT $ORACLE_HOME/RECOVERY/etc	VARIABLE
REQUIRED SETTINGS:		
Database Name	**ORACLE_SID**	*ORACLE_SID*
Default *mt*, *tar* device	**DEFAULT_TAPE**	*TAPE*
OPTIONAL SETTINGS:		
Full backup/restore device	**FULL_TAPE**	*FULL_TAPE*
Redo backup/restore device	**REDO_TAPE**	*REDO_TAPE*
Cumulative backup/restore device	**CUM_TAPE**	*CUM_TAPE*
Alternate print destination	**PRINTER**	*PRINTER*

D.3.3.1.6 Preliminary Checking. The preliminary checking function is designed to test for certain program initialization conditions and will not allow the backup process to continue if they exist. The following program initialization conditions are checked:

- Not logged in as <*oradba*> — The program cannot write output files or perform any SQL*DBA functions.
- A backup lock is found — Program cannot start if another backup or restore program is currently in progress.
- *ORACLE_SID* or *TAPE* — The program cannot perform any SQL*DBA "is null" functions or cannot use any *mt* or *tar* commands.

D.3.3.1.7 Exit Handling. The exit handling function is designed to handle three types of exits. The first type is controlled by the program flow. The second type occurs as a result of a system failure, and the third type of exit is initiated by the user via the TERMINATE FULL BACKUP option from the BACKUP STATUS MENU.

In all of the above exit modes, the exit handling function is responsible for removing lock and temporary files. When a fatal system/program error occurs, the exit handler performs the additional function of ending the tablespace backup. The additional function is needed to insure that the program does not exit with a tablespace altered to begin backup.

The exit mode functionality is accomplished through two Unix *trap* commands. The first *trap* command is triggered by an exit signal 0 and performs the cleanup functions. The second *trap* command is triggered by an emergency exit which is characterized by the following signals:

Table D-3: Full Backup Program Emergency Exits.

SIGNAL	MEANING AND TYPICAL USE
1	Hangup — stop running. Sent when a user runs the **terminate full backup** program.
2	Interrupt — stop running. Sent when a user types: <*KEYBOARD()*>.
5	Quit — stop running. Sent when a user types: <>.
9	Kill — stop immediately. Emergency use.
15	Optional signal which can be used to terminate cleanly if possible.

The second *trap* calls the **end_ts_backup** module which ends the tablespace backup. For details on **end_ts_backup** see Copying Database Files to Tape (Paragraph D.3.3.3.2). It is important to note that the first *trap* will always run and is not dependent on the second *trap* being triggered.

D.3.3.2 Tape Verification Process. The tape verification process is designed to execute a series of tape identification operations when a full backup tape is placed in the drive preventing the user from accidently overwriting a cumulative or redo log tape or some other tape. It keeps track of the correct tape sequence (**FULL_1**, **FULL_2**) and will not overwrite a **FULL_1** tape when the **FULL_2** tape is required. The identify tape routines are incorporated into the **identify_btape** external module.

D.3.3.2.1 identify_btape — External Module. identify_btape is called from the full and cumulative backup programs when the first tape is inserted and for each subsequent tape required to complete the backup. It verifies the state of the tape drive (e.g., checks if a tape inserted) and the **FULL_n** or **CUM_n** tape type, and prompts the user for a particular action if a verification fails. **identify_btape** is self contained and performs the same function for both the full and cumulative backups.

ARGUMENTS:

- First Argument — *CUM $TAPE_CNT/FULL $TAPE_CNT* — The type of backup (full or cumulative) and current sequence (1..N). Compares this parameter with name of the first file on tape to determine if they are the same.
- Second Argument — Program Initialization Directory Path — Directory where the **br_back_full** or **br_back_cum** resides. The path is used to locate the **current_progress** file when writing the status messages.
- Third Argument — Application Name — The application name "Full Backup" or "Cumulative Backup" used in the status and error messages.

RETURNED EXIT STATUS:

0 Tape verification was successful: a *tar* read error was encountered indicating a new tape. In both cases the **created_header** module is called.

1 Tape verification was unsuccessful: This could result because a redo log backup tape or other tape was in the tape drive. It will also result from a *tar* error because the tape drive was busy or because the verification process has timed out after approximately three hours.

identify_btape is called from the main program body as part of the *if* command conditional expression. This allows the calling program to test the returned exit status. It uses the returned exit status to determine that the correct tape header was found. If **identify_btape** returns a 0, then it found the correct tape. If it returns a nonzero, **identify_btape** encountered an error and the calling program initiates the error exit routines. **identify_btape** is driven by an unconditional *while* loop construct. It will break out of the loop if it finds the correct header, encounters an error, or is timed out (approximately three hours) without finding the correct tape. When a user action is required, the message prompts are repeated at set intervals of 60 seconds. An example of this condition would be insertion of the wrong tape; the backup cannot continue until the **FULL_1** tape is inserted. The messages repeat until the required action is completed or a maximum time interval of approximately three hours is met. For a detailed description of messages see User Prompting/**send_msg_to_term** Module, Paragraph D.3.3.1.4.

The module identifies the tape under three categories: 1) rewinding/no tape inserted; 2) at the beginning; or 3) not positioned at the beginning of the tape. It handles each category differently.

- Rewinding/no tape inserted — creates an offline message: wait and try again.
- At the beginning of the tape — reads the first file and identifies it as a **FULL** or **CUM** header. If the tape is identified, rewind to the beginning then return. If it cannot be

identified, then it displays the appropriate message and continues looping. If a *tar* error is encountered, then it returns and tries to create the header in case it is a new tape.

- Not positioned at the beginning of the tape — checks to see if the device is busy. If it is, then it handles the error and returns an error exit status. If the device is not busy, it rewinds the tape and restarts the verification operation.

D.3.3.2.2 create_header Module. The **create_header** module creates a new header on a blank tape or overwrites the old header with a new one. The newly created header file is used by the verification process. The header file **(FULL_N)** is overwritten because the date information in the header file is updated each time a header is required on tape.

The **create_header** module is called after the **identify_btape** returns a successful exit status. It creates a file **FULL_1..N** in the runtime directory containing the current date. The module then copies the file to tape. The same error handling applies as described in Paragraph D.3.3.3.2, Copying Database Files to Tape/**end_ts_backup** Module

D.3.3.3 Database File Backup Process. The backup process is comprised of the following functions:

- Identifying Database Files, Paragraph D.3.3.3.1
- Copying Database Files to Tape/**end_ts_backup** Module, Paragraph D.3.3.3.2
- Processing Multitape Backups, Paragraph D.3.3.3.3.

D.3.3.3.1 Identifying Database Files. This function performs the task of identifying all available database files and their tablespace counterparts. The list of database files and tablespaces are derived from the **DBA_DATA_FILES** table. The **DBA_DATA_FILES** table is a data dictionary view and provides a built in source for finding the current database files.

The tablespace name and database files selected from the **DBA_DATA_FILES** have a two-fold purpose:

- Supply the backup process with a file containing an entry for each database file along with the corresponding tablespace.

- Provide the backup process with a file containing all the tablespaces and database files which will act as a look-up table so the program can find the database files associated with a tablespace.

ORACLE procedures — Identify Database Files

The set of commands which make up the identifying function are part of the main body of the **br_back_full** script. The procedures to identify the database files are accomplished using ORACLE and Unix commands and are as follows:

C SQL*DBA — Query **DBA_DATA_FILES** Table Data

Select the **tablespace_name** and **file_name** along with an **OK_ROW** constant and redirect the standard output to **/tmp/spool**. The **OK_ROW** identifies the legitimate rows so they can be removed later.

C Error Handling— ORACLE (Type— Terminating)

If an attempt to query the **DBA_DATA_FILES** table results in errors, they will be redirected to the **/tmp/spooled** file and identified. The **/tmp/spooled** file is then passed as an argument to the error handler and processed.

D.3.3.3.2 Copying Database Files to Tape/end_ts_backup Module. The copying database files to tape function is designed to accommodate online serial database backups. Online backups require that each tablespace must be altered to begin backup before the datafiles can be copied to tape. In addition, the tablespaces must also be altered to mark the end of the tablespace backup after the files are backed up. This process must be performed for each tablespace until all the database files are copied to tape. A tablespace iterator is used to identify and prepare tablespace and database file information so it can be used in the *tar* command in a sequential manner.

The set of commands which compose the copying function are part of the main body of the **br_back_full** script. The process of copying the database files to tape can be divided into the following two stages:

C Determine the tablespaces designated for backup and assign to an iteration variable.

- Implementation is as follows:

  For 1 to N iterations

  $TABLESPACES ----------> *TSNAME* variable

- Extract unique tablespace names and assign to the iteration variable:

  /tmp/input -------> *TABLESPACES* variable

C Process each tablespace as per the tablespace iterator as follows:

- Prepare the ORACLE tablespace for backup.

- Group the database file names and locations by tablespace and store that information to a file.

- *tar* those database files to tape using the file with a *-I* option so the *tar* command can locate the those files.

- Signal the end of the ORACLE tablespace backup.

The **end_ts_backup** module contains the SQL*DBA commands to mark the end of each tablespace backup. The commands are grouped into a module because it is used in several places throughout the application.

Error Handling — ORACLE

The output of the SQL*DBA command is filtered for any occurrences of ORACLE error messages and redirected to a temporary **error_flag** file. Interpretation of the **error_flag** file is as follows:

Empty — SQL*DBA command was successful.

Not empty — An ORACLE error occurred. The **error_flag** is passed to the error handler and processed.

Error Handling — *tar*

The standard error from the *tar* command is redirected to a variable. Interpretation of the standard error variable is as follows:

Null — *tar* command was successful.

Not Null/Terminating — If the value does not match "blocksize = 0" or "unexpected End of File (EOF)" then *tar* was not successful. Pass the standard error variable to the error handler.

Not Null/Tape Control — If the value matches "blocksize = 0" or "unexpected EOF", then the end of recordable media has been reached. The backup process requires an additional tape in sequence to continue. The tape sequence is incremented and the backup process continues. For detailed design information see Processing Multitape Backups, Paragraph D.3.3.3.3.

D.3.3.3.3 Processing Multitape Backups. This function is designed to automatically handle multitape backups. A multitape backup occurs when additional tapes are required to complete the backup process. When the end of tape occurs the process must be temporarily suspended to allow the user to insert the next tape. Each additional tape is verified via the tape verification process.

An additional tape is required when *tar* returns "blocksize = 0" or "unexpected EOF" which indicates that the end of recordable media has been reached. The function responds by incrementing the tape counter and calling **identify_btape** external module with the incremented tape name, e.g., **FULL_2**. When the correct tape is verified, the new header is copied to tape via the **create_header** module. The backup process begins with the file it was copying to tape when the additional tape was required.

D.3.3.4 Auxiliary File Backup Process. The postbackup process performs additional backup functions after the database files are copied to tape. It spools the **V$FILESTAT** table information to the **vfilestat** file and backs up the ORACLE **control**, **init.ora** file, and database structure. The **V$FILESTAT** information is the current number of blocks written to each tablespace file. The cumulative program uses this information to determine which tablespaces have been modified since the last full backup. The backup control file, **init.ora** file, and exported database structure are copied to tape. Also, the program marks the tape with a special "end of backup" EOB file. The EOB file allows the restore program to identify the last tape so it can automatically exit.

The backup **control** and Database structure export file were extracted from the database via the *ALTER DATABASE* and *exp* commands. The auxiliary files are bundled and copied to tape. The program spools out database file I/O information from the **V$FILESTAT** table to a **vfilestat** file.

Error Handling — ORACLE

If an attempt to query the **V$FILESTAT** table results in errors, the errors will be redirected to the **vfilestat** file and identified. The **vfilestat** file is then passed as an argument to the error handler and processed.

Error Handling — *tar*

See Paragraph D.3.3.3.2, Copying Database Files to Tape, for error handling details.

D.3.3.5 Status Report Process/create_exit_log Module. The status report process provides the user an overall backup summary. If the status is successful or successful with warnings, then the report will contain the database files copied to tape grouped by tablespace name. A terminated status report will include the errors that caused the interruption of the backup program. The status report is sent to the printer. In the case of a terminated error the status report is also mailed to the <*oradba*> and <*secman*> users.

This module is called from the main program if any critical stage of the backup encounters a fatal error and cannot continue or if the program completes the backup successfully. This module evaluates the input arguments and determines the status. This module categorizes the backup status into three possibilities:

- SUCCESSFUL - Backup and Postbackup process completed. All database and auxiliary files were copied to tape.

- SUCCESSFUL - Backup process completed. Postbackup process encountered WITH an ORACLE or *tar* terminal error. All database files were WARNINGS copied to tape so it is not necessary to repeat backup.

- TERMINATED - Tape verification or Backup process encountered a fatal ORACLE, *tar*, or Unix error. The backup must be repeated.

INPUT ARGUMENTS:

- First Argument — Indicates main status category (success or terminated).

- Second Argument — Further defines the success status as completing with a no warning or warning status.

D.3.4 Cumulative Backup — br_back_cum. br_back_cum performs an online cumulative database backup of all tablespaces in which updates, inserts, or deletes were applied since the last full backup. The recovery of all database files will require only the full and most recent cumulative tape. This will minimize down time. All datafiles of each tablespace are backed up while the database is in use for normal operation. The commands and modules which comprise the cumulative backup program are grouped together to form the following processes:

- Program Initialization Process — handles program execution/termination, user/program interaction and preliminary checking. For details see Paragraph D.3.3.1, Program Initialization Process.

- Tape Verification Process — performs identification of cumulative backup tapes and enforces the correct type of tape and the correct sequence (**CUM_1**, **CUM_2**, etc.). For details see Paragraph D.3.3.2, Tape Verification Process.

- Database File Backup Process — determines which tablespaces have changed and locates the name of each tablespace and its corresponding database files from the **DBA_DATA_FILES** table and copies those files to tape. For details see Paragraph D.3.3.3, Database File Backup Process.

- Auxiliary File Backup Process — copies the backup control **init.ora** and Database structure export files to tape. For details see Paragraph D.3.3.4, Auxiliary File Backup Process.

- Status Report Process — compiles and prints a backup summary log which contains the backup status (successful, successful with warnings, and terminated with the error message). Also, if the program was successful it contains the database files copied to tape grouped by tablespace name. The file is printed out at the end of the backup so the user can place the printout in the backup binder. For details see Paragraph D.3.3.5, Status Report Process.

INPUT TABLES:

- **SYS.DBA_DATA_FILES** — provides the tablespace names and database file names.

- **SYS.V$FILESTAT** — provides number of physical blocks written to each tablespace file.

OUTPUT FILES:

- **vfilestat** (current and since last full backup) — contains the spooled out **V$FILESTAT** information.

- **current_progress** — gives the user the ability to check the progress of the program. The file contains messages which describe all program events taken place so far. The **current_progress** will also contain any program errors.

- **exit_status** — gives the user a backup summary log which contains the status of the program (successful, successful with warnings, terminated with error message). Also, if the program was successful it contains the files copied to tape grouped by tablespace name. The file is printed out at the end of the backup so the user can place the printout in the backup binder.

- **error.log** — an accumulation of all error messages.

TEMPORARY FILES:

- **/tmp/spooled** — output from the **SYS.DBA_DATA_FILES** table.

- **/tmp/ts_list** — contains the tablespace names and database file names which are derived from the spooled file. This file is the main driver of the program.

- **/tmp/error_flag$$** — this file holds any ORACLE errors which are processed by the error handler.

- **/tmp/LK_BACKUP_CUM$$** — created when the program first starts and locks out any other instances of the backup or restore program from running simultaneously.

- **/tmp/TABLESPACE_NAME** (e.g., **/tmp/CARRIER**) — contains all database files that make up the tablespace. For each tablespace to be backed up, one **TABLESPACE_NAME** file is created. For instance, **CARRIER** tablespace will have a file named **CARRIER** which contains **/home10/sm1/carrier1.dbf**.

- **/tmp/inclause** — contains tablespace names used in the where clause for selecting data from the **DBA_DATA_FILES** table.

GENERAL ERROR HANDLING:

All routine errors are trapped by the **handle_error** external module, including write/permission errors, lock errors (another backup/restore program is currently running), *tar* I/O errors, and login errors. Refer to Paragraph D.3.2, External Error Handler Module — **handle_error**, for more details.

When any of the above errors occur, the program writes the appropriate error message to the **error_log**, **current_status**, and **exit_status** files, then the error message is mailed to the <*oradba*> and <*secman*> user accounts.

Most of the errors will result in the unrecoverable termination of the program. If an error occurs while backing up the control file or while the **V$FILESTAT** table is being accessed the program will terminate successfully with a warning. At this point, there is no need to restart the backup because all the files are already backed up to tape.

D.3.4.1 Database File Backup Process. The backup process is comprised of following functions:

- Finding Changed Tablespaces, Paragraph D.3.4.1.1
- Identifying Database Files, Paragraph D.3.4.1.2
- Copying Database Files to Tape, Paragraph D.3.4.1.3
- Processing Multitape Backups, Paragraph D.3.4.1.4.

D.3.4.1.1 Finding Changed Tablespaces. The set of commands which make up this function are part of the main body of the **br_back_cum** script. The **br_back_cum** program uses the **V$FILESTAT** information from the full backup and the recently spooled **V$FILESTAT** information from the cumulative backup. The **V$FILESTAT** information contains the number of blocks written to each database file. The number of blocks written to each database file at the time of the full backup are compared with the current number of block writes. The tablespaces are selected for backup if the number of block writes have changed since the last full backup. The **V$FILESTAT** information for the full backup is spooled to **$ORACLE_HOME/RECOVERY/backup/full/vfilestat** and cumulative backup is spooled to **$ORACLE_HOME/RECOVERY/backup/cum/vfilestat**.

If the program cannot determine the tablespace I/O activity since the last full backup, (e.g., cannot find a full backup **vfilestat** file) it performs a full backup. The program backs up each tablespace sequentially by using the *ORACLE BEGIN BACKUP* and *END BACKUP* commands before and after it copies the tablespace files to tape.

If no database files have changed since the last full backup, the program will indicate that no changes to the database have occurred since the last backup.

The **br_back_cum** program performs a *diff* on those **vfilestat** files and extracts the tablespace name and number of blocks written. The output of the *diff* command is the main driver of the program. The output is processed and provides a list of tablespaces slated for backup. The program then places the tablespace list in a *where* clause in order to get the tablespace file names from the **DBA_DATA_FILES** table. **br_back_cum** generates a list of files from the information extracted from the **DBA_DATA_FILES** table. Each file is named after a tablespace and contains the tablespace database file name and path. The generated files are used in the *tar* command as part of the *-I* option. As each file which identifies the tablespace and database files is generated, **br_back_full** prepares the tablespace, copies the database files to tape, then moves to the next tablespace.

D.3.4.1.2 Identifying Database Files. Same function as the full backup. For details see Paragraph D.3.3.3.1, Identifying Database Files.

D.3.4.1.3 Copying Database Files to Tape. Same function as the full backup. For details see Paragraph D.3.3.3.2, Copying Database Files to Tape/end_ts_backup Module.

D.3.4.1.4 Processing Multitape Backups. Same function as the full backup. For details see Paragraph D.3.3.3.3, Processing Multitape Backups.

D.3.5 Redo Log Backup Program — br_back_redo

The **br_back_redo** program maintains the offline archive storage area by preventing the archive storage area from becoming full. The archive storage area is critical to the operation of ORACLE. When ORACLE can no longer archive the online redo log files, it stops all access to the database, until the excess redo log files are deleted.

br_back_redo is designed to keep the archive storage area clear by monitoring the archive directory storage capacity and moving the excess redo logs to tape or to designated contingency drives. Under normal circumstances **br_back_redo** will move the archived redo logs to tape automatically when archive disk usage is greater than or equal to 50 percent. If **br_back_redo** is locked out by another backup program and archive disk usage is greater than or equal to 95 percent, it will move the redo logs to contingency drives. The commands and modules which comprise the redo log backup program are grouped together to form the following processes:

- Program Initialization Process — handles program execution/termination, user/program interaction, and preliminary checking.

- Tape Verification Process — performs identification of the redo log tape and verifies the beginning redo log sequence number on tape. It enforces the type of tape and makes sure that the same tape is used for successive redo log backup until the tape is full. This is critical because the numerical order of redo logs on tape cannot be out of sync.

- Redo Log Backup Process — locates any redo logs residing in the archive default and contingency directories. The backup function then copies the redo logs to tape and removes them from the archive and contingency directories.

- Contingency Process — locates all contingency file systems and first qualifies available file systems by determining if there is enough free space to store 10 redo log files. The contingency function sorts available file systems by most available free space then relocates the redo log files to these contingency areas. The process continues until the disk usage of the archive redo log file system is reduced to less than 50 percent, or all contingency file systems are processed.

- Contingency Space Usage Process — checks the combined average disk usage of all ORACLE contingency file systems. This function is performed after all contingency file systems are processed. If the average contingency disk usage is greater than or equal to 95 percent, then the full or cumulative backup program preventing **br_back_redo** from running is killed and the appropriate messages are sent.

- Status Report Process — compiles and prints a backup summary log which contains the backup status and also lists the ranges of redo log sequence numbers copied to tape and the starting redo log sequence of the first redo log stored on tape. The file is printed out at the end of the backup so the user can place the printout in the backup binder. The starting sequence redo log number helps the user to determine which tape is required to restore the proper redo log files.

INPUT FROM ORACLE:

ARCHIVE LOG LIST command (optional) — provides the ORACLE default archive path. The archive path is written to a file which makes the *ARCHIVE LOG LIST* command optional. If ORACLE crashes, the **br_back_redo** program can still function by retrieving the archive path from the backup file.

OUTPUT FILES:

- **archive_dest** — is accessed if ORACLE is unavailable and provides the archive log path.

- **last_seq_num** — contains the last sequence redo log number copied to tape. It is used to verify the starting redo log file. If the program finds any redo logs with sequence numbers less than the **last_seq_num**, then those redo log files will not be included in the *tar* list.

- **REDO_LOG** — header file containing the date and the starting redo log sequence number on tape. The header file is used to identify the type tape. The starting redo log sequence number is used during the restoration process. If the sequence redo log

number that ORACLE requests is less than the starting redo log sequence number on tape, then it is the wrong tape.

- **current_progress** — gives the user the ability to check the progress of the program. The file contains messages which describe all program events which have been completed. The **current_progress** will also contain any program errors.

- **exit_status** — gives the user a backup summary log which contains the exit status of the program (successful, successful but encountered errors, and terminated with the error message). In addition, if the program was successful, it contains the range of redo log sequence numbers moved to tape and the ranges of redo log sequences that were out of order. The **exit_status** produces a printout of the above information and the user can place the printout in the backup binder.

- **error.log** — is an accumulation of all error messages.

TEMPORARY FILES:

- **/tmp/br_err_msg** — all error messages are initially written to this file which is then output to the **current_progress** and **exit_status** file.

- **/tmp/LK_BACK_REDO$$** — locks out any other instance of the backup or restore program.

GENERAL ERROR HANDLING:

All routine errors are trapped by the **handle_error** external module including write/permission errors, lock errors (another backup/restore program is currently running), *tar* I/O errors, and login errors. Refer to Paragraph D.3.2, External Error Handler Module **handle_error**, for more details.

When any of the above errors occur the program writes the appropriate error message to the **error_log, current_status**, and **exit_status** files. All error messages (with the exception of lock error messages) are mailed to the <*oradba*> and <*secman*> user accounts.

A detailed description is provided in the following paragraphs.

D.3.5.1 Program Initialization Process. The program initialization process is comprised of the following functions:

- Program Execution, Paragraph D.3.5.1.1
- Program Termination, Paragraph D.3.5.1.2
- Monitoring Current Status, Paragraph D.3.5.1.3
- User Prompting, Paragraph D.3.5.1.4
- Environment Variable Declaration, Paragraph D.3.5.1.5

- Runtime Variable Declaration, Paragraph D.3.5.1.6
- Preliminary Checking, Paragraph D.3.5.1.7
- Exit Handling, Paragraph D.3.5.1.8.

D.3.5.1.1 Program Execution. The modes of execution include the BACKUP AND RECOVERY MENU and the **cron** batch processing facility. The **br_back redo** program works most effectively, when executed from **crontab**, because it is designed to monitor and move redo logs with no interaction from the user, other than insertion of the correct tape. The program runs as a background process regardless of the execution mode. By default, the **cron** facility executes the backup programs in background. In the menu execution mode the *nohup* command, in conjunction with the *&* background operator, places the menu selected program in background which gives the user immediate control of the terminal. As a result, the user can then log off the terminal or continue working with no affect on the backup process.

Processing steps are dependent on the source of initiation:

- Initiated from **cron**: It moves offline redo log files to tape automatically. The program checks the disk usage at every interval of time as set in the <*oradba*> user **crontab** file. When the archived redo log storage area becomes greater than or equal to 50 percent full, all archived redo log files are copied to tape and removed from the hard drive.

- Initiated manually from BACKUP: It moves all archived redo log files to tape when the BACKUP AND RECOVERY menu option is selected, regardless of the space used in the archived redo log storage area.

The **br_redo_back** program can determine if it is initiated from **cron** or the menu by the argument used at the command line (*br_back_redo CRON* or *br_back_redo MENU*).

When no other backups are currently in progress, the **br_back_redo** program performs the typical operations of moving the excess redo logs to tape. When **br_back_full** or **br_back_cum** locks out the **br_back_redo** program, and archived redo log disk usage is greater than or equal to 95 percent, the **br_back_redo** program performs the concurrent operation of moving the excess redo logs to contingency redo log drives.

D.3.5.1.2 Program Termination. The termination option gives the user the ability to stop the full backup at any time. Program termination is initiated via the TERMINATE FULL BACKUP option from the BACKUP STATUS MENU.

D.3.5.1.3 Monitoring Current Status. The monitoring function and implementation is the same as in the full backup program. For details see Paragraph D.3.3.1.3, Monitoring Current Status.

D.3.5.1.4 User Prompting. The redo log backup implementation of sending messages to screen is similar to the full backup implementation, except the **REDO_LOG** tape label is used instead of the

FULL_1 tape label. For details see Paragraph D.3.3.1.4, User Prompting/**send_msg_to_term** Module.

D.3.5.1.5 Environment Variable Declaration. The full and redo log backup implementation of the environment variable declaration function are similar. For Environment Variable Declaration implementation details see Paragraph D.3.3.1.5.

D.3.5.1.6 Runtime Variable Declaration. The redo log backup program requires certain ORACLE and Unix information before it can perform a backup. **br_back_redo** assigns this information to runtime variables at the beginning of the program.

The values are derived from the *ARCHIVE LOG LIST*, ORACLE and Unix system commands. The various runtime variables are described below:

ARCHIVE_PATH Contains the full redo log archive directory path. Used to determine the location of the archive redo logs within the default archive directory.

ARCH_FORMAT Contains the name of the redo log files without the sequence number or file extension (e.g., GCCS_1234.log -> GCCS). Used as a format filter when searching the contingency drives to distinguish archive redo log files from other types of files.

MOUNT_POINT Contains the first level of the archive directory path (e.g. /oracle/smback/arch/GCCS—> /oracle/smback). The first level directory is the mount point of the archive file system. The archive mount point is used to determine disk usage, using the *df* command.

ARCH_USAGE Contains the disk usage of the archive directory mount point /oracle/smback, derived using the *df* space utilization command. It is the percent used value. The *ARCH_USAGE* value is evaluated to determine if redo logs should be moved to tape (greater than or equal to 50 percent), or moved to contingency drives (greater than or equal to 95 percent).

ARCH_CURR_SEQ Contains the current online redo log sequence number. Used to determine which redo log will be the last one copied to tape. The order is from the oldest to the most recent redo log.

D.3.5.1.7 Preliminary Checking. The full and redo log backup implementation of the environment variable declaration function are similar. For details see Paragraph D.3.3.1.6, Preliminary Checking.

D.3.5.1.8 Exit Handling/cleanup Module. The **br_back_redo** and **br_back_full** exit handling functionality are similar in the use of the *trap* command. The **br_back_redo** *trap* command calls the **cleanup** module, while the **br_back_full** *trap* command calls the **end_ts_backup** module. The

cleanup module provides the routines to enforce an orderly exit which will not be affected by an unexpected premature exit condition. An orderly exit is required because the next time the **br_back_redo** program runs it must continue the process of copying the next redo log to the same tape. The **cleanup** module performs the general duties of synchronizing the program sequence number and the last redo log sequence number copied to tape.

The **cleanup** module is used in the command line of the first *trap* and is called when a normal exit is completed. The second *trap* is triggered when an unexpected exit occurs, and it communicates its status to the **cleanup** module by setting the *EXIT_STATUS* flag to "2". After the second *trap* is triggered, the first *trap* will run, and the cleanup module will be called with an *EXIT_STATUS* of "2". When a normal exit is completed, the *EXIT_STATUS* retains the original program setting which can be a "0" or a "1". The **cleanup** module interprets exit status and, depending on the value, performs the functions shown in Table D-4.

Table D-4: Redo Log Backup Program Exit Status Flags.

EXIT_STATUS	MEANING/CLEANUP ACTION
0 or 1	The program completed a normal exit.
2 with *tar* error	Unexpected exit. The current tape could be damaged. The program encountered an error while copying a file to tape. Take the appropriate measures to override the tape verification process which will allow the user to use a different tape next time **br_back_redo** is executed.
2 with no *tar* error	Unexpected exit. Determine if any redo log files were copied to tape. If so, extract the last file copied to tape and compare the redo log sequence number to the value store in the **last_copied_seq** file. When the redo log sequence numbers are the same, then the program and tape redo log sequence numbers are in sync. When the redo log sequence number on tape is greater than the **last_coped_seq** sequence, then the redo log file was copied to tape but not removed from the hard drive. The redo log must be deleted from the drive or it will be recopied to tape when the program starts again.

The **cleanup** module is executed every time **br_back_redo** exits, but the *EXIT_STATUS* value will determine which action is taken.

D.3.5.2 Tape Verification Process. The tape verification process is designed to implement the necessary steps to ensure that redo log files are copied to a redo log tape in successive order until the tape becomes full. For example, the redo log backup program copies all redo logs up to redo log sequence 90 to tape during one session. When the next session starts, the program needs to copy redo logs 91, 92, and 93 to the same tape. The tape verification process not only verifies the redo log tape but must also ensure that redo log 91 is copied to the tape which contains redo log 90. The sequence of redo logs must be maintained or the redo log restore program will not work. The tape verification process also prepares a new tape with a **REDO_LOG** header. The **identify_tape** and **create_header** modules are used to perform the tape verification process.

D.3.5.2.1 identify_tape Module. The **identify_tape** module executes the verification routines which prevent the user from accidentally overwriting a cumulative or full tape. It also keeps track of the redo log sequence order on tape.

The **identify_tape** module is called from the main module to process the first tape and from the **exec_tar_process** module to process subsequent tapes. The **identify_tape** module is included as part of the *while* command conditional expression. This allows the main program to test the returned *EXIT_STATUS*. It uses the returned *EXIT_STATUS* to determine if the correct tape header was found. If the module returns a "1", then it found the correct tape and the *while* loop is terminated. If it returns a "0", the main program initiates the **send_msg_to_term** module and continues looping calling the **identify_tape** module. It will break out of the loop if it finds the correct header, encounters an error, or times out (approximately three hours) without finding the correct tape. The message prompts are repeated at set intervals of 60 seconds when a user action is required.

For example, if a full backup tape is inserted, the backup cannot continue until the correct redo log backup tape is inserted. The messages are repeated until the required action is completed or a maximum time interval of approximately three hours is reached. For a detailed description of messages see Paragraph D.3.3.1.4, User Prompting/**send_msg_to_term** Module.

INPUT ARGUMENTS:

- First Argument — Describes the type of backup and is used when messages are displayed.
- Second Argument — Called from the main module. Tells the **identify_tape** module to continue appending logs on current redo log tape.

INITIAL_PASS:

- Second Argument: Called from **exec_tar_process**. Tells the **identify_tape** module that the current tape is full. The module will allow the user to start the redo log copy process with a new tape or a previously used tape.

OUTPUT PARAMETERS:

1 Terminate **identify_tape** *while* loop:

- Indicates that the **identify_tape** process verified the tape type and starting log redo sequence number.
- The module encountered a *tar* error. Before the module returns a "1" it creates a new **REDO_LOG** file header on tape and on the hard drive.

0 Continue the **identify_tape** *while* loop.

C Indicates that the inserted tape is a full or cumulative backup tape.

C The tape has a REDO_LOG file header but the log redo sequence numbers on tape are out of sync.

The **identify_tape** procedures are designed to handle each of the following three situation:

C **Situation One. No Tape Loaded—** Create an offline message, wait, and try again.

C **Situation Two. At the Beginning of Tape:**

- **identify_tape** called with *INITIAL_PASS* Argument. In this situation the verification process enforces the continuation of redo logs on the same tape currently in use for redo log backups. The program will not allow the user to change tapes until the current redo backup tape is full. The procedures to accomplish this are as follows:

  -- Extract the beginning redo log sequence number from the header on tape. Verify that the start sequence from the header on tape matches the starting redo log sequence number from the header on the hard drive. If the redo log sequence numbers match, move the tape forward to the end of recorded media and return a "1" to terminate the *while* loop. The redo logs will be appended to the end of the tape. If the redo log sequence numbers do not match then return a "0" and display the wrong tape message and restart the verification operation.

  -- The **identify_tape** module will call the **create_header** module if the tape is new or *tar* error occurs.

- **identify_tape** called with *NEXT_TAPE* Argument. In this situation the verification process enforces the continuation of redo logs on a new tape or a previously used redo log backup tape. The program will not write redo logs on the currently used tape because it is full. The procedures to accomplish this are as follows:

  -- Extract the beginning redo log sequence number from the header on tape. Verify that the start redo log sequence from the header on tape does not match the starting redo log sequence number from the header on the hard drive. If the redo log sequence numbers do not match, rewind the tape and return a "1" to terminate the *while* loop, then call the **create_header** module. If the redo log sequence numbers are the same, then return a "0" and prompt the user to insert the next tape. The **identify_loop** will continue.

-- The **identify_tape** module will call the **create_header** module if the tape is new or a *tar* error occurs.

C **Situation Three. Not at The Beginning of Tape** - In this case the module will attempt to identify the last file written to tape to see if it is a redo log file. If it is a redo log file, the tape goes to the end and returns. If it is not a redo log file, it rewinds the tape and displays a wrong tape message. This represents normal operation. The user leaves the tape in the drive and, whenever the redo log backup program kicks off, it continues to write on the same tape until it is full. If any other conditions exist (*tar* error, etc.,), it will place an error flag in the **REDO** file header on the hard drive, so the user can use a different redo backup tape on the initial run.

D.3.5.2.2 create_header Module. The **create_header** module creates a new header on a blank tape or overwrites the old header. The header file **(REDO_LOG)** contains the date and redo log sequence number of the first redo log written to tape. The starting redo log sequence number is used by the **identify_tape** module in its verification process.

D.3.5.3 Redo Log Backup Process. The main objective of the backup process is to locate redo logs residing in the archive default and contingency directories, copy those redo logs to tape and then remove them from the source directories. The backup process accomplishes that objective by completing the following three functions:

C Locating the Archive Redo Logs — search all file systems owned by the <*oracle*> user and compile a list of redo log files.

C Processing Redo Log Sequence Numbers — sort the redo log sequence number portion of the redo log file name and verify the starting and ending redo log sequence number.

C Moving Logs to Tape — verify the redo log sequential order. If out of order, keep track of the missing range of redo log sequence numbers then copy the **tar** file to tape.

D.3.5.3.1 Locating the Archived Redo Logs. Locating the archived redo logs is the first stage in the backup process. The **br_back_redo** program calls the **get_arch_file_list** module which performs this task. The **get_arch_file_list** module searches all mounted file systems owned by the <*oracle*> user. It uses the *$ARCH_FORMAT*_[0-9]*.log pattern to qualify the redo log files. The ARCH_FORMAT variable is initialized with the name of the redo log files without the redo log sequence number or file extension (e.g., SM_1234.log -> SM). All redo log files that match the pattern are assigned to the *ARCH_FILE_LIST* variable.

D.3.5.3.2 Processing Redo Log Sequence Numbers. This stage of the backup process is required because the archived redo log sequence is critical. The archived redo logs are copied to tape in sequential order from the oldest (smallest) sequence to the most recent (largest) sequence. The sequential order must also be applied to archived redo logs located on contingency drives which must be merged and sorted along with redo logs located in the archived redo log default directory.

The sequential order is maintained by processing the redo log sequence numbers. The sequence numbers are stripped from the redo log files assigned to the *ARCH_FILE_LIST* variable. See Locating the Archive Logs, Paragraph D.3.5.3.1, for more information on the *ARCH_FILE_LIST* variable. Once the redo log sequence numbers are sorted in ascending order, all redo log sequence numbers which are older (less) than the last redo log sequence copied to tape are removed. This insures that, in the event a redo log file was copied to tape and not removed, it will not be recopied to tape. The processed redo log sequence numbers are assigned to the *SEQ_NUM* variable. The first redo log sequence number in the list is assigned to the *OLDEST_SEQ* variable. The *MOST_RECENT_SEQ* variable is derived by decreasing the current online redo log sequence number by one. The list of sequence numbers along with the *OLDEST_SEQ* and *MOST_RECENT_SEQ* variables are used in copying the redo log files to tape.

D.3.5.3.3 Moving Redo Logs to Tape. Moving the archived redo logs to tape is the final stage of the backup process. At this point the redo log sequence numbers are in order, and boundaries for the oldest and the newest redo log sequence numbers are established. This information is used to process each redo log file before they are copied to tape by determining the starting redo log sequence and verifying the sequential order. If gaps are found in the redo log sequential order, then the program alerts the user to the missing range of redo log sequence numbers. To accomplish this task, the program calls **tar_arch_logs** and **exec_tar_process** modules. The **br_back_redo** program calls the **tar_arch_logs** module which starts the process. The **tar_arch_logs** module calls the **exec_tar_process** when a redo log file is ready to be copied to tape.

D.3.5.3.3.1 tar_arch_logs Module. This module is called from the main module after the archive redo log list is generated, and the redo log sequence numbers are sorted. The list of redo log sequence numbers drives a *for* loop construct. Each redo log sequence, from a complete list of redo log files, is compared to a CURRENT_SEQ sequence which is initially set to the last copied sequence plus one. The CURRENT_SEQ sequence is used to detect any redo log file sequences that are missing so that the user can be aware that a gap exists between redo logs. The *CURRENT_SEQ* sequence is normally incremented by one and if each redo log sequence from the list is equal to the CURRENT_SEQ value then the redo logs are in order. If at any point the redo log sequence does not equal the CURRENT_SEQ, the gap is noted and the *SEQUENCE* is then set to the redo log sequence that was out of order and the process continues. In any case, the redo log file is copied out to tape and removed.

The CURRENT_SEQ sequence number is used to look-up the redo log file name and path in the list of redo logs generated by the **get_arch_file_list** module. Once the redo log file name is found, the directory path is separated from the redo log file name and both values are used as input parameters to the **exec_tar_process** module. At that point the files are copied to tape.

D.3.5.3.3.2 exec_tar_process Module. The **exec_tar_process** is called from the previous module **tar_arch_logs**. **exec_tar_process** performs the *tar* operation for **tar_arch_logs**. It copies the *tar* file to tape excluding the directory path (allowing the restore program to extract the *tar* files to the default oracle archive directory). It then removes the redo log file from the hard drive and returns to the calling module. The **exec_tar_process** manages the end of tape operation and calls the **identify_tape** with the *NEXT_TAPE* argument. (See Paragraph D.3.5.2, Tape Verification Process, for more information on **identify_tape**.)

INPUT ARGUMENTS:

- First Argument — The redo log file name. It is used as part of the *tar* command.
- Second Argument — The full path. It is used to locate and remove the redo log file from the hard drive.

D.3.5.4 Contingency Process. The main objective of the contingency process is to relocate the excess redo log files from the archive directory to the available contingency file systems. This situation occurs when the archived redo log storage area becomes greater then or equal to 95 percent full and the redo log backup program is locked out by a full or cumulative backup. The contingency process completes the following functions:

- Finding Available Contingency Drives — finds all available <*oracle*> user mount points and sorts them by free disk space (greatest amount to the least).
- Relocating Excess Redo Log Files — relocates excess archive redo log files to available <*oracle*> user file systems until the disk usage of the archive redo log file system is reduced to less than 50 percent or until all contingency file systems are used.

D.3.5.4.1 Finding Available Contingency Drives/arch_move_logs Module. The first phase of finding available contingency drives is sorting all mounted file systems by the availability of free space in descending order. This helps reduce fragmentation over time. The second phase involves the process of qualifying each mounted file system owned by the <*oracle*> user under the following criteria:

- The mounted file system must have enough free space to hold at least ten archived redo log files. The redo logs are moved in batches of ten between each available archived free space check because the *df* command is **cpu** intensive (i.e., it should not be used after each redo log is moved).
- The mounted file system cannot be *ORACLE_HOME* or the archived redo log default directory.

The **arch_move_logs** module accomplishes the task of locating available contingency drives. It is called from the main module after the archived log list is compiled and the redo log sequence numbers are sorted. The loop is driven by the space usage output of a processed *df* command. If a

file system meets the criteria of the second phase described in the paragraph above, then the **arch_move_logs** module calls the **find_and_move** module and executes the relocation operation.

D.3.5.4.2 Relocating Redo Log Files/find_and_move Module. This stage completes the contingency process and works in conjunction with the **arch_move_logs** module. After the **arch_move_logs** module secures a contingency file system it calls the **find_and_move** module. The **find_and_move** module performs the operation of moving redo logs to contingency drives.

The **find_and_move** module uses a sorted listing of archived redo logs. The listing is processed by a loop construct which provides the driver for moving each redo log into the contingency directory. The module moves redo logs in batches of 10 redo logs and checks archive redo log disk usage. The **find_and_move** module continues moving redo logs until archive redo disk usage is less than 50 percent or until the contingency file system is full. It then returns the current archived redo log directory usage to the calling module.

The **arch_move_logs** module evaluates the returned value. If the value is greater than or equal to 50 it continues to call the **find_and_move** module. If the value is less than 50, it terminates the relocation process.

D.3.5.5 Contingency Space Usage Process/check_free_space Module. The contingency space usage process provides an emergency plan of action when all the following conditions exist:

- All contingency file systems are full
- The redo log backup program is locked out
- Free archived redo log storage space has reached critically low levels.

In this case the relocation of redo logs is not an option. The only available option is to terminate the backup program that is preventing the redo log backup from running.

The contingency space usage process accomplishes this by calling the **check_free_space** module and evaluating the returned value. The **check_free_space** module returns the combined average disk usage of all <*oracle*> user contingency file systems. If the average contingency disk usage is greater than or equal to 95 percent, the error handler is called. The full or cumulative backup program preventing **br_back_redo** from running is killed and the appropriate messages are sent.

The **check_free_space** module is called from the main program after contingency process is finished. It uses the output of the *df* command to calculate the combined average disk usage. This value is returned to the main program were it is evaluated.

D.3.5.6 Status Report Process. The redo log and full backup exit status reports have the same function. The only difference is the output of a successful report. The redo log exit report prints a range of redo log sequences that were copied to tape. It also alerts the user to ranges of redo logs that were not found. See Paragraph D.3.3.5, Status Report Process, for more details.

D.3.6 Full/Cumulative Restore Programs — br_restore_full/br_restore_cum The **br_restore_full** and **br_restore_cum** programs share the same functionality. The **br_restore_full** program restores all the database files from the full backup tape and the **br_restore_cum** program restores all the database files from the cumulative backup tape. As each program extracts files from tape, the program keeps track of multitape restore sessions and prompts the user for the next tape in sequence. Both the **br_restore_full** and **br_restore_cum** programs are designed to restore the database files by completing the following four processes:

Program Initialization Process — deals with program termination, user/program interaction and preliminary checking.

Tape Verification Process — performs identification of full or cumulative backup tapes and enforces the correct type of tape and the correct redo log sequence.

Database File Restore Process — extracts the files from tape so they can be restored to the original directory paths.

Status Report Process — compiles and prints a backup summary log which contains the backup status (successful, successful with warnings, and terminated with the error message). If the program was successful, it will contain the database files copied to tape grouped by tablespace name. The file is printed out at the end of the backup, and the user can place the printout in the backup binder.

OUTPUT FILES:

exit_status — gives the user a backup summary log which contains the exit status of the program (successful, successful but encountered errors, and terminated with the error message). In addition, if the program was successful, it will contain the range of redo log sequence numbers moved to tape and the ranges of redo log sequences that were out of order. **exit_status** produces a printout of the above information and the user can place the printout in the backup binder.

error.log — an accumulation of all error messages.

TEMPORARY FILES:

- **/tmp/LK_RESTORE_FULL$$** : Created when the program first starts and locks out any other instance of the backup or restore program from running concurrently. The file is removed when the program terminates via the *trap* command and is removed from the **/tmp** directory automatically if the system is rebooted.

- **/tmp/LK_RESTORE_CUM$$**: Same as the **LK_RESTORE_FULL$$**.

- **/tmp/br_err_msg$$**: All error messages are initially written to this file which is then output to the **current_progress** and **exit_status** file.

- **/tmp/error_flag$$**: Contains the output of the SQL*DBA command and is used to determine the state of the database (closed or online). Contains the results of connecting to ORACLE and will contain errors because the database is not online when performing a restore.

- **/tmp/restore.list**: Contains the current list of database files restored from tape. The list is kept up to date by appending the restored file name to the **restore.list** file each time the *tar* command is executed. The contents of the file are then displayed to screen so the restore process can be monitored.

GENERAL ERROR HANDLING:

- All routine errors are trapped by the **handle_error** external module, including write/permission errors, lock errors (another backup/restore program is currently running), and *tar* I/O errors. Refer to Paragraph D.3.2, External Error Handler Module **handle_error** for more details.

- When one of the routine errors described above occurs, the program writes the appropriate error message to the **error.log**, **current_status** and **exit_status** files. Then the error message is mailed to the <*oradba*> and <*secman*> user accounts.

There are two types of errors: terminal and nonterminal. A terminal error causes the program to output the errors and exit. A nonterminal error causes the program to output the errors and continue. Terminal errors are for login violations, file I/O errors and any *tar* I/O errors.

D.3.6.1 Program Initialization Process. The program initialization process is comprised of the following functions:

- User/Program Interaction, Paragraph D.3.6.1.1
- Environment Variable Declaration, Paragraph D.3.6.1.2
- Preliminary Checking, Paragraph D.3.6.1.3
- Emergency Error Handling, Paragraph D.3.6.1.4.

D.3.6.1.1 User/Program Interaction. The **br_restore_full** and **br_restore_cum** programs are executed from the BACKUP AND RECOVERY MENU. This gives the user the option to quit or continue at certain stages of the restore process or under certain conditions. The program prompts the user and waits for a response to continue or, in some cases, gives the user the option to quit. The restore programs require minimum user interaction. Each program displays the current progress, exit status report, and any errors encountered to the screen. In addition, the **exit_status** is printed.

D.3.6.1.2 Environment Variable Declaration. It is important to note that the *TAPE*, *ORACLE_HOME*, and *ORACLE_SID* environment variables are initialized in the **br_main_menu** program and exported to the restore programs. The **br_restore_full/cum** program initialization derives the environment declaration and error checking from the **br_main_menu** program.

D.3.6.1.3 Preliminary Checking. The **br_restore_full/cum** program initialization derive the preliminary checking from the **br_main_menu** program which has the same function as performed by the full backup program. See Paragraph D.3.3.1.6, Preliminary Checking, for details.

D.3.6.1.4 Emergency Error Handling. An emergency exit occurs as a result of a system failure or unexpected program termination. Emergency exits are handled via a *trap* command which is responsible for removing temporary and the lock files. The signals which trigger the *trap* command are shown in Table D-5.

Table D-5: Full/Cumulative Restore Emergency Error Handling.

SIGNAL	MEANING AND TYPICAL USE
1	Hangup— stop running. User selects the TERMINATE FULL BACKUP option from the BACKUP STATUS MENU.
2	Interrupt— stop running. Sent when a user types <^C>.
5	Quit— stop running. Sent when a user types <^\>.
9	Kill— stop immediately. Emergency use.
15	Optional signal which can be used to terminate cleanly if possible.

D.3.6.2 Tape Verification Process. The tape verification process performs the following functions:

- Restore Preparations, Paragraph D.3.6.2.1
- Tape Verification/**identify_rtape** — External Module, Paragraph D.3.6.2.2.

D.3.6.2.1 Restore Preparations. Restore preparations include disabling the <*oradba*> user **crontab** file and shutting down the database. The <*oradba*> user **crontab** file is disabled to prevent a scheduled backup from executing while in the process of restoring files from tape. The database is *shutdown* to preserve the integrity of the database files.

D.3.6.2.2 Tape Verification/identify_rtapes — External Module. The identify_rtapes external module is designed to identify the tape type and track of the correct tape sequence (**FULL_1**, and **FULL_2**). The identification of the tape type and sequence insures that the Full Restore program will restore files from the **FULL_1** tape and will not allow the user to accidently restore files from a **FULL_2**, **CUM_1**, or some other tape.

The **identify_rtape** external module is called from the full and cumulative restore programs after the user acknowledges the prompt for the first tape and for each tape required to complete the restore process. It verifies the state of the tape drive (e.g., determines if a tape is inserted) and the **FULL_n** or **CUM_n** tape type then prompts the user for a particular action if a certain verification fails. **identify_rtape** is self-contained and performs the same function for both the full and cumulative backups. See Paragraph D.3.3.2.1, **identify_btape** External Module, for details.

ARGUMENTS:

- First Argument — *CUM $TAPE_CNT/FULL $TAPE_CNT* — the type of restore (full or cumulative) and current sequence (1..N). Compares this parameter with the name of the first file on tape to determine if they are the same.
- Second Argument — Program initialization Directory Path — directory where the **br_back_full** or **br_back_cum** resides. The path is used to locate the **current_progress** file when writing the status messages.
- Third Argument — Application Name — the application name "Full Restore" or "Cumulative Restore" which is used in the status and error messages.

RETURNED EXIT STATUS:

- 0 — Tape verification was successful. The tape was identified.
- 1 — Tape verification was unsuccessful. This may result if a tape is out of sequence, if a redo log backup tape or some other tape is in the drive, or from a *tar* error. It may also result if the tape drive is busy or the verification process has timed out after approximately three hours.

The **identify_btape** and **identify_rtape** implementation strategies are very similar except that **identify_btape** identifies a tape to be written to, and the **identify_rtape** identifies a tape to be read from. The differences are in the way **identify_rtape** handles an unknown *tar* error condition and the position of the tape when it returns successfully. **identify_rtape** terminates the program when it encounters a *tar* error while the **identify_btape** returns and tries to create a header file on the tape. **identify_rtape** returns with the tape positioned after the header file and at the beginning of the first database file while the **identify_btape** rewinds the tape to the beginning before it returns.

D.3.6.3 Database File Restore Process. The restore process performs the task of restoring each database file from tape to the original directory path, (displaying a current list of restored files), and determining if additional tapes are required to complete the restore process. These tasks are performed for each database file restored from tape.

The main component of the restore process is the *tar* command. As the *tar* command restores the files from tape, the output of the command containing the file name is redirected and appended to the **restore.list** file. The **restore.list** file contains the current list of restored files and is displayed to screen after each execution of the *tar* command. The restore process determines if additional tapes are required by searching for the **EOB** file at the end of the tape after all the files on tape are restored. If the **EOB** file is found, the restore process rewinds the current tape and terminates the process. If the **EOB** file is not found, the restore process suspends the current operation, prompts

the user for the next tape, performs the tape verification process, then continues with the restore process.

D.3.6.4 Status Report Process/create_exit_log Module. The status report process provides the user with a summary of the overall restore status. If the status is successful, the report will contain the database files restored from tape. A terminated status report will include the errors which caused the interruption. The status report is displayed on the screen and will remain on the screen until the user acknowledges the report by pressing <*return*>. This insures that the user will not miss the results of the restore program. The status report is then sent to the printer.

The module is called from the main programs when a fatal error is encountered and the program cannot continue or when the restore program completes successfully. The module evaluates the input arguments and determines the status. The module categorizes the status into two possibilities:

- SUCCESSFUL — The tape verification and restore processes completed all operations. All database files were restored from tape.
- TERMINATED — Tape verification or restore process encountered a fatal ORACLE, *tar*, or Unix error. The restore operation must be repeated.

INPUT ARGUMENTS:

- First Argument — Indicates the status category (success or terminated).
- Second Argument — The current runtime directory which the status report uses when writing out the report to a file and locating the **restore.list** file.
- Third Argument — Indicates the calling application name (Full Restore or Cumulative Restore). The name is used in the status report header.

D.3.7 Redo Log Restore Program — br_restore_redo

The **br_restore_redo** program runs after the Full and Cumulative restore programs and restores the database up to the point of failure. It acts as an interface to the ORACLE recovery process by managing the redo logs required to restore the database. The program also works with ORACLE in applying those archived redo logs.

The program extracts only the required archived redo logs from tape and searches the contingency drives for additional redo logs required for the recovery of the database. **br_restore_redo** works in conjunction with ORACLE SQL*DBA to determine if any redo logs are required from tape. The **br_restore_redo** program is comprised of the following processes:

- Program Initialization Process — handles program termination, user/program interaction, and preliminary checking.

- Restore Preparation Process — performs required preliminary tasks (database *shutdown*, disabling **cron,** and running SQL*DBA) in order to prepare for the tape verification process.

- Tape Verification Process — performs identification of the redo log backup tape. It enforces the correct tape type and verifies that the required starting redo log sequence number is stored on tape. In addition, the tape verification process locates the specific starting redo log on tape before the restore process takes place.

- Redo Log Restore Process — extracts the redo log files from tape so they can be restored to the original directory paths.

- Redo Log Location Report Process — compiles and prints a report which shows the location of redo logs within each contingency drive and the ORACLE commands required to apply the redo logs to the database.

- Status Report Process — compiles and prints a report which shows the overall status of the program (successful or terminated because of an error). In addition, if the program was successful, it shows the range of redo logs sequences restore from tape.

OUTPUT FILES:

archive_dest — contains a backup copy of the archive redo log default directory path. Normally, **br_restore_redo** derives the archived redo log directory path from the *ARCHIVE LOG LIST* command. If ORACLE is unavailable, the **br_restore redo** program uses the **archive_dest** value.

last_copied_seq — contains the last redo log sequence number copied to tape. Used to verify the starting redo log file. If the program finds any redo logs with sequence numbers less than the **last_copied_seq**, then those redo log files will not be included in the *tar* list.

current_progress — gives the user the ability to check the progress of the program. The file contains messages which describes all program events taken place so far. The **current_progress** will also contain any program errors.

exit_status — gives the user a backup summary log which contains the exit status of the program (successful, successful but encountered errors, and terminated with the error

message). In addition, if the program was successful, it will contain the range of redo log sequence numbers moved to tape and the ranges of redo log sequences that where out of order. The **exit_status** produces a printout of the above information, and the user can place the printout in the backup binder.

error.log — an accumulation of all error messages.

TEMPORARY FILES:

/tmp/br_err_msg — all error messages are initially written to this file which is then output to the **current_progress** and **exit_status file**.

/tmp/REDO_LOG — header file from tape containing the date and starting log sequence number. The header file is used to identify the tape and to verify that the starting sequence number on tape is greater than the sequence number that is required to begin recovery.

/tmp/LK_RESTORE_REDO$$ — locks out any other instances of the backup or restore program from running concurrently.

GENERAL ERROR HANDLING:

- All routine errors are trapped by the **handle_error** external module, including write/permission errors, lock errors (another backup/restore program is currently running), and *tar* I/O errors. Refer to Paragraph D.3.2, External Error Handler Module **handle_error,** for more details.

- When one of the routine errors described above occurs, the program writes the appropriate error message to the **error.log**, **current_status** and **exit_status** files. Then the error message is mailed to the <*oradba*> and <*secman*> user accounts.

D.3.7.1 Program Initialization Process. The program initialization process is comprised of the following functions:

- User/Program Interaction, Paragraph D.3.7.1.1
- Environment Variable Declaration, Paragraph D.3.7.1.2
- Preliminary Checking, Paragraph D.3.7.1.3
- Emergency Exit Handling, Paragraph D.3.7.1.4.

D.3.7.1.1 User/Program Interaction. The **br_restore_redo** program is executed from the BACKUP AND RECOVERY MENU. Restore programs selected from this menu run in program mode. The user has the option to quit or continue at certain stages of the restore process whenever the user prompts are displayed. The program displays the current progress, Log Location Report, Status Report, and any errors encountered to the screen.

D.3.7.1.2 Environment Variable Declaration. It is important to note that the *TAPE*, *ORACLE_HOME*, and *ORACLE_SID* environment variables are initialized in the **br_main_menu** program and exported to the restore programs. The **br_restore_redo** program initialization process derives the environment declaration and error checking from the **br_main_menu** program.

D.3.7.1.3 Preliminary Checking. The **br_restore_redo** program initialization process also derives the preliminary checking from the **br_main_menu** program which has the same functionality as the full backup. See Paragraph D.3.3.1.6, Preliminary Checking, for details.

D.3.7.1.4 Emergency Exit Handling. An emergency exit occurs as a result of a system failure or unexpected program termination. Emergency exits are handled via a *trap* command which is responsible for removing temporary and locked files. The signals which trigger the *trap* command are shown in Table D-6.

Table D-6: Redo Log Restore Emergency Error Handling.

SIGNAL	MEANING AND TYPICAL USE
1	Hangup— stop running. Sent when a user selects the TERMINATE FULL BACKUP option from the BACKUP STATUS screen.
2	Interrupt— stop running. Sent when a user types: <*KEYBOARD()*>.
5	Quit— stop running. Sent when a user types: <>.
9	Kill— stop immediately. Emergency use.
15	Optional signal which can be used to terminate cleanly if possible.

D.3.7.2 Restore Preparations Process. Restore preparations include disabling the <*oradba*> user **crontab** file, shutting down the database, and running the SQL*DBA program. The program functions in the following sequence:

- The <*oradba*> user **crontab** file is disabled to prevent a scheduled backup from executing when the program is restoring files from tape. This is critical because the redo log backup contingency process is designed to move redo logs from the archive directory when disk usage is greater than or equal to 95 percent. It is possible, while restoring redo logs from tape, that the archive disk usage could exceed 95 percent if **crontab** is not disabled. In this situation, it is possible that restored log files could be moved to contingency drives before being applied to the database. This would cause a delay in database recovery.

- The database is *shutdown* to preserve the integrity of the database files.

- The program provides access to SQL*DBA because the initial process of applying the redo logs to the database is accomplished through ORACLE. When the user enters the <*ALTER DATABASE RECOVER*> command at the SQL*DBA prompt, ORACLE searches the archive directory for the redo log sequence which matches the

oldest missing database changes. The results of the *RECOVER* command determine if ORACLE found the required redo log files in the archive directory, or if the starting redo logs are on tape. If the *RECOVER* command completes successfully, then the recovery process is complete. If the *RECOVER* command senses a missing sequence, then the restore process required additional redo logs from tape. If it cannot find the starting redo log sequence, it displays the missing redo log sequence number. The user is required to enter the redo log sequence number in order for the program to continue to the tape verification process. The missing redo log sequence is the starting redo log sequence for redo logs required from tape, up to and including the last redo log sequence copied to tape.

D.3.7.3 Tape Verification Process. The tape verification process performs the following functions:

- Tape Verification, Paragraph D.3.7.3.1
- Tape Positioning/**find_seq_log** Module, Paragraph D.3.7.3.2.

D.3.7.3.1 Tape Verification/identify_tape Module. The tape verification function performs the task of identifying the tape type and verify that the required starting redo log sequence is on tape. The function will only accept tapes with the **REDO_LOG** header as the first file. Once the tape is accepted, the first sequence on tape is compared with the missing starting redo log sequence required to recover the database. The tape verification function will only accept redo log tapes where the first redo log sequence on tape is greater than the missing starting redo log sequence. This is a time saving feature. The tape verification function also prompts the user for a particular action if verification fails.

The **identify_tape** module is called from the following:

- Main Body: Calls **identify_tape** after the user acknowledges the prompt for the first tape.

- **extract_log** Module: **extract_log** restores redo log files from tape and calls the **identify_tape** module when redo logs from an additional tape are required to complete the restore process.

- **find_seq_log** Module: **find_seq_log** searches the redo log backup tape for the starting redo log and will call **identify_tape** when it encounters the end of tape without finding the redo log.

The **identify_tape** module is called as part of the *if* command conditional expression. This allows the calling module to test the returned exit status. It uses the returned exit status to determine that the correct tape header was found. If the subprogram returns a "0" then it found the correct tape. If it returns other than "0", the calling module initiates the exit routines and terminates with an error.

INPUT ARGUMENTS:

- First Argument: Describes the type of backup and is used when messages are displayed.

RETURNED EXIT STATUS:

- 0 Tape verification was successful. A **REDO_LOG** file header was found and the starting redo log sequence is on tape.

- 1 Tape verification was unsuccessful which could result from a full/cumulative backup tape or some other tape, a *tar* error, or tape drive was busy.

The **identify_tape** module is designed to reinitiate the verification process if it encounters an empty tape drive or the wrong tape. It accomplishes this by incorporating the verification tests in a *while* loop construct. The *while* loop is broken if the correct tape is found or by user request (via message prompts). The *while* loop will continue until the correct tape is found, the drive is busy or a *tar* error is encountered. The **identify_tape** module performs the following verification tests:

- No Tape Loaded — create an offline message and wait for user acknowledgement. Start again or quit as per user input.

- At the Beginning of Tape — extract the redo log sequence number from the header on tape. This redo log sequence number identifies the first redo log on tape. The extracted redo log sequence number must be greater then the redo log sequence number required to start the recovery process. If the extracted redo log sequence number is less then the starting sequence, the user is alerted and the verification process continues with the next inserted tape.

- Not at the Beginning of Tape — determine if the drive is busy. If it is busy, then handle the error and break out of the loop. If the drive is not busy, then rewind the tape and restart the verification function.

D.3.7.3.2 Tape Positioning/find_seq_log Module. The tape positioning function performs the task of reading each redo log on tape until it finds the redo log on tape which is required to begin the recovery process. If the redo log is found, the tape will be properly positioned to begin the restore process.

The **find_seq_log** module is called after the **identify_tape** has successfully verified the initial redo log backup tape. It uses a *tar tvb* and *grep* command to move the tape forward while searching for the matching redo log sequence number. When the module reads the matching redo log sequence, the tape is moved forward to the next redo log file. The tape must be repositioned to the beginning of the previous matching redo log.

D.3.7.4 Redo Log Restore Process. The restore process manages the various operations involved in restoring the required redo log files from tape to the archive redo log default directory. It keeps track of available space in the archive redo log directory while redo logs are being extracted from tape. When free space on the archived disk becomes critically low, it suspends the restoration of redo logs to the database and enters SQL*DBA. The user now has the opportunity to apply those extracted redo logs to ORACLE. Upon exiting SQL*DBA, those redo logs are removed from the archive redo log directory. The restore process accomplishes these operations by using the following functions:

- Extracting Redo Logs from Tape, Paragraph D.3.7.4.1
- Applying the Restored Redo Logs/**rm_log_scrn** Module, Paragraph D.3.7.4.2.

D.3.7.4.1 Extracting Redo Logs from Tape. This function is comprised of a two-fold operation which includes extracting log files from tape to the ORACLE default archive redo log directory and keeping track of the available archive redo log directory space while the redo logs are being restored. The two-fold operation makes best use of the available archive redo log directory space by restoring redo log files until the space is full. The redo logs are then applied to the database and removed so the space can be reused. If all the redo logs are restored from the starting redo log sequence to last redo log sequence copied to tape, then the space management operation is not necessary.

D.3.7.4.1.1 space_avail Module. The **space_avail** module controls the number of redo logs restored to the archive redo log directory. The **space_avail** module controls the flow of redo logs through construction of the *while* conditional expression (e.g., *while* **space_avail**). For each redo log restored, the **space_avail** module is called in order to determine the current disk usage for the archive redo log directory. Depending on the returned status of the **space_avail** module, the *while* loop will continue or break out. The called module will return a "1" if disk usage is greater than or equal to 95 percent or, if the last redo log restored, is the last redo log sequence required from tape to complete the recovery.

D.3.7.4.1.2 extract_log Module. The **extract_log** module works in conjunction with the **space_avail** module. For each iteration of the *while* **space_avail** loop, the **extract_log** module is called. The *tar* command is used to extract each redo log.

D.3.7.4.2 Applying the Restored Redo Logs/rm_log_scrn Module. When the *while* **space_avail** inner loop is broken, the restored redo log is applied. This function uses the SQL*DBA facility to apply restored redo logs to the database and then removes those redo logs automatically. The apply and remove operations continue until all required redo logs from tape are applied to the database. This is accomplished by an outer loop which tests the most recently restored redo log sequence. The outer loop is broken when the restored redo log sequence equals the last redo log sequence copied to tape. As long as the most recently restored redo log sequence is less than the last redo log sequence copied to tape, the inner *while* **space_avail** loop is executed and the restored redo logs will be applied.

The **rm_log_scrn** is called after the restored redo logs are applied to the database via the SQL*DBA facility. The module displays a screen showing the beginning and ending redo log

sequence numbers of applied redo logs. The screen also displays a warning indicating that those redo logs must be removed. When the user acknowledges the message, the redo logs are removed from the archive redo log directory. If the user decides not to remove the redo logs, the program is terminated.

D.3.7.5 Redo Log Relocation Report Process. The redo log relocation report process is initiated after the restored redo logs from tape are applied to the database. The process manages the operation of applying redo logs from the contingency drives. Contingency drives act as a redo log contingency drive overflow area. The process first determines if any contingency redo logs exist by searching all contingency drives. If any redo logs are found, a redo log contingency report is produced, containing the range of redo logs on each contingency drive and the SQL*DBA command needed to apply those redo logs.

D.3.7.5.1 conting_log_notfound Module. The **conting_log_notfound** module determines if any redo logs where moved to contingency drives. To accomplish this, the module isolates the sequence number of the oldest redo log in the archive redo log directory /oracle/smback/arch and compares it to the last redo log sequence copied to tape. If the oldest redo log sequence in /oracle/smback/arch is the next redo log required to recover the database, then there is no gap between the logs on tape and the logs under /oracle/smback/arch. In this case, no logs are located on contingency drives. For example, if the last redo log sequence copied to tape is 100, and the oldest redo log sequence in /oracle/smback/arch is 101, no logs are located on contingency drives. In this case the module will return a "0" and the program will create the status report and exit.

Using a similar example, if the last redo log sequence copied to tape equals to 100 and the oldest redo log sequence number in the archive directory is 107, then sequences 101 to 106 are located on the redo log contingency drives. When contingency redo logs are found, the module returns a "1" and the log relocation report is compiled.

D.3.7.5.2 get_arch_file_list Module. The **get_arch_file_list** module searches all mounted file systems owned by the <*oracle*> user. It uses the *$ARCH_FORMAT*_[0-9]*.log pattern to qualify the redo log files. The *ARCH_FORMAT* variable is initialized with the name of the redo log files without the sequence number or file extension (e.g., SM_1234.log -> SM). All log files that match the pattern are assigned to the *ARCH_FILE_LIST* variable. The logs in the *ARCH_FILE_LIST* variable are sorted then formatted (for example, directory_path:log file) and the results are written to the **log.done$$** file. An additional file **arch_dir$$** containing a list of directories is also created. Both files are used to create the redo log relocation report.

D.3.7.6 Status Report Process. The status report process provides the user with a summary of the overall restore status. If the status is successful, the report will contain the range of restored redo log sequences. A terminated status report will include the errors which caused the interruption. The status report is displayed on the screen and will remain on the screen until the user acknowledges the report by pressing <*return*>. This insures that the user will not miss the results of the restore program. The status report is sent to the printer.

D.3.7.7 create_exit_log Module. This module is called from the main program when a fatal error is encountered and the program cannot continue or when the restore program completes successfully. The module evaluates the input argument and determines the status. The module categorizes the status into two possibilities:

- SUCCESSFUL — All required redo log files were restore from tape.
- TERMINATED — Tape verification or restore process encountered a fatal ORACLE, *tar*, or Unix error. The restored operation must be repeated.

INPUT ARGUMENT:

- First Argument Indicates the status category (0 SUCCESS or 1 TERMINATED).